Reusable Software Components[†]

Bruce W. Weide, William F. Ogden, and Stuart H. Zweben

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210-1277

1. Introduction

For over twenty years programmers have dreamed of developing software systems that are engineered like traditional physical systems. The vision of a software industry resembling the

[†] To appear in *Advances in Computers*, M.C. Yovits, ed., Academic Press, 1991.

piece-part component industries supported by electrical and mechanical engineering is generally credited to (McIlroy, 1976) in remarks at a 1968 NATO conference. Much as their counterparts use standard integrated circuits, gears, or nuts and bolts, software engineers might design and build programs from standard *reusable software components*. A similar view based on “software ICs (integrated circuits)” has been popularized more recently by (Cox, 1986).

Despite certain progress in this direction, most software systems are not developed from reusable components. In part this is because some software engineers have not accepted component-based software technology as a worthwhile goal. Moreover, for many years we have been urged by influential people (e.g., professors) to use “top-down” analysis and design. Software design based on reusable components demands more than a modicum of “bottom-up” thinking throughout the development cycle.

1.1 Electronics Engineering vs. Software Engineering

A simple example illustrates the difference in viewpoint between the electronics and software industries. Suppose a customer approaches a representative of ABC Electronics Company with a request for a chip implementing a 13-bit adder circuit. The reply is likely to be, “We’ll be happy to sell you two 8-bit adders that can be cascaded to do 16-bit addition. That will probably meet your needs, and we can deliver tomorrow at a bargain price.” Only a particularly ambitious (or naive) salesperson might continue, “Now I’m sure our engineers can develop a 13-bit adder if that’s what you really want. It will cost you a lot more and take six months, though.” The response is based on the observation that this customer’s apparent requirements do not match precisely what the company has to sell. Historically, the default view in electronics engineering has been that *requirements* should be adapted to fit what off-the-shelf components are able to do. The electronics salesperson offers a variant of what the customer asks for.

Faced with an analogous request from a customer for a piece of software, a representative of XYZ Software Company is likely to reply, “No problem! There’s no customer need we can’t meet.” The response is based on the same observation about the customer’s apparent requirements. The difference, of course, is that historically the default view in software engineering has been that software is by definition “soft” and can be made to do virtually anything. *Programs*, not customer requirements or expectations, are what should be adapted. The software salesperson offers to do a custom job.

Either point of view might be sensible, depending on the circumstances. Indeed the electronics and software design communities appear to be moving toward a middle ground. For example, the former has been relying more — but far from exclusively — on custom and semi-custom VLSI circuits for critical but special-purpose applications (Gu and Smith, 1989). The fact that software systems should be designed using a combination of top-down and bottom-up thinking has also become more widely recognized (Fairley, 1985).

Certainly there are cases where software should be considered as a malleable, highly customizable commodity just as it has been in the past. In many other circumstances, though, the desired characteristics should be generality, reusability, and parameterizability. How to achieve these qualities in software components is the focus of this chapter.

There are, of course, well-documented non-technical impediments to adopting the electronics engineering metaphor for software even where it is clearly appropriate: management resistance, incentive problems, psychological barriers such as the “not invented here” syndrome, and many others. This chapter does not consider such issues, which have been surveyed elsewhere, e.g., (Tracz, 1987; Biggerstaff and Perlis, 1989). Frankly, there are not too many specific advances to report on this front in any case. There also remain surprisingly serious technical impediments to taking advantage of reusable software components. Study of these problems is important because without solutions to them, all the best intentions of managers, programmers, and users cannot make reuse a reality (Booch, 1987; Meyer, 1987; Tracz, 1987; Meyer, 1988). Furthermore, in the technical arena specific progress can be identified.

1.2 Objective and Overview

The primary objective of this chapter is to review and consolidate important recent technical advances that should help make reusable software components more routinely feasible in the future. A secondary objective is to identify serious technical problems that demand further attention in order to be solved satisfactorily. A reader with modest background and experience in software design and development should have no trouble following the presentation and appreciating the issues.

There are so many approaches to software reuse that all could not possibly be discussed in a single chapter, nor could even the major issues all be covered adequately. Fortunately, there are recent and accessible treatments of the overall area that paint a nice broad-brush view, such as (IEEE, 1984; Biggerstaff and Perlis, 1989). One indisputable fact that emerges from the reuse literature is that *software must be designed for reuse* (Tracz, 1990a). The focus here is therefore on the current state of the art and practice relative to the *design* of reusable software components. We concentrate on two main questions:

- How can the behavior of reusable software components be explained so a user can understand *what* it does without knowing *how* it does it?

- What specific guidelines can help a software engineer to design components that are highly reusable?

The organization of the presentation is as follows. Section 2 establishes a framework and terminology in which the above questions can be phrased more precisely. It starts with the description of a general model of software structure, then uses that model to clarify several key ideas, including “software component.” The model leads to a natural vision of the likely scope of a mature software components industry, similar in many respects to that postulated by McIlroy, and illuminates the technical issues that are considered in greater detail in Sections 3, 4, and 5.

Section 3 reviews recent work on software specification, i.e., description of the behavior of software that respects the fundamental principles of information hiding and abstraction. Section 4 discusses several general guidelines that have been proposed as rules-of-thumb in design for reuse and introduces some more specific corollaries that are useful to a designer of practical components. Section 5 briefly considers two other important questions: parameterization of behavior and certification of correctness of component implementations.

It is impossible to separate completely questions about the design of reusable components from the specification and implementation languages used to express those designs. It is equally impossible in the space of this chapter to survey in depth the plethora of languages that incorporate features that might be considered to support software reuse. We therefore concentrate on a few recent languages that potentially could serve as the basis for a future software components industry. These include Ada (Ada, 1983), C++ (Stroustrup, 1986), and Eiffel (Meyer, 1988). Our own REusable SOftware Language with Verifiability and Efficiency, a.k.a. RESOLVE (Hegazy, 1989; Harms, 1990; Sitaraman, 1990), consolidates recent advances toward reusable components into a uniform framework and includes what we have found to be the best features of other languages for this purpose. The dialect of RESOLVE used in this chapter reflects the ideas of the language but is slightly more compact than the full syntax, which is designed to support more complex designs than can be discussed in the available space. The forerunners and relatives of these languages include Simula-67 (Birtwistle *et al.*, 1973), Alphard (Shaw, 1981), CLU (Liskov *et al.*, 1981), Smalltalk (Goldberg and Robson, 1983), Objective-C (Cox, 1986), and many others.

2. Framework

The purpose of this section is to consider a component-based reuse technology for software engineering that parallels that of the traditional engineering disciplines. We conclude that in the future an economically significant portion of all software development activity will fall into the realm of a software components industry. The remainder of the chapter addresses issues related to development of a mature industry of this kind.

2.1 A Model of Software Structure

Fig. 1 shows how we consider a software system to be structured. The circles denote *abstract components* and the rectangles denote *concrete components*. A thin arrow from an abstract component to a concrete component means the latter *implements* the former. A thick arrow from a concrete component to an abstract component means the latter is *directly used* by the former. Throughout the chapter we occasionally consider this small window on a much larger software system from the point of view of a representative concrete component called "Client."

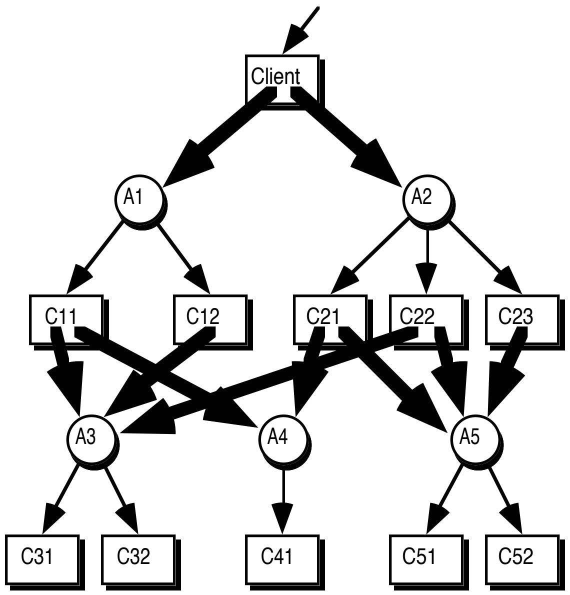

FIG. 1. Software Model with Abstract (A) and Concrete (C) Components.

There are many possible interpretations of the words in the previous paragraph, and the figure makes sense for just about any of them. In fact, it makes sense as a model of general system structure even outside the software arena. We choose to give the figure a particular meaning: A *component* is (in programming language terms) a module, package, or class, typically providing an abstract data type and associated operations. It comprises an abstract part to explain behavior along with a concrete part to implement that behavior. In the previous paragraph these two pieces are called abstract and concrete components, respectively, because it is important to distinguish between them and to consider them as separate but related entities for most purposes.

For example, suppose Client is a piece of code that uses stacks and queues. Then A1 might be a stack abstraction and A2 a queue abstraction. C11 might be code that implements stacks by representing a stack as an array with a top index, while C12 implements the same abstract interface by representing a stack as a list of nodes. They key idea — by now well-known to software engineers (Parnas, 1972) — is that in order to *program* Client one should need to understand only the abstract components describing the visible behavior of stacks and queues (i.e., A1 and A2). In order to *execute* Client, however, one also must choose concrete components that implement stacks and queues (i.e., one of C11 or C12, and one of C21, C22, or C23). In fact, one must choose concrete components implementing these concrete components' abstract constituents, and so forth down through the hierarchy to the machine level.

As a result of working group discussions at the Reuse in Practice Workshop in July 1989, a common vocabulary has been proposed to facilitate discussion of issues related to software having this kind of structure (Tracz and Edwards, 1989; Tracz, 1990a). Subsequent elaboration by the members of the group at the Workshop on Methods and Tools for Reuse in June 1990 (Latour *et al.*, 1990; Edwards, 1990; Tracz, 1990b) has given the "3C reference model" the potential to become the accepted basis for discourse on reusable software components among members of the reuse community. We adopt the model here in order to support this trend toward a much-needed common intellectual and terminological framework.

The 3C model defines and distinguishes three ideas:

- *Concept*: a statement of what a piece of software does, factoring out how it does it; abstract specification of functional behavior.

- *Content*: a statement of how a piece of software achieves the behavior defined in its concept; the code to implement a functional specification.

- *Context*: aspects of the software environment relevant to the definition of concept or content that are not explicitly part of the concept or content; additional information (e.g., mathematical machinery and other concepts) needed to write a behavioral specification, or additional information (e.g., other components) needed to write an implementation.

For variety we use the terms "concept" and "abstract component" interchangeably; similarly, "content" and "concrete component." The notion of "context" is a bit harder to pin down because there is separate context for concept and content. Section 5.1 clarifies the idea and suggests how different aspects of context can be separately related to concept and content.

2.2 Reusable Software Components

The 3C model of software structure makes no commitment as to whether any component is reused. Fig. 1 illustrates reuse because there are abstract components with multiple thick arrows directed inward (indicating that an abstract component is directly used by more than one concrete component), but it is possible to envision a tree-structured diagram rather than the directed acyclic graph shown here. The model also makes no commitment as to the source of the components, i.e., whether they are purchased piece-parts or leftovers from the last company project or developed as custom components for the current project. The model thus provides a framework in which reusable components can be studied but it does not mandate reuse.

Furthermore, among the vast array of techniques for reuse and the artifacts that could be reused, the model makes no judgments about which should be reused. Various authors emphasize reuse of high-level "domain analysis" across several entire projects within an application area (Prieto-Diaz, 1990). Others concentrate on "code" reuse, but there is no general agreement on precisely what this means. Some envision components being reused by explicit composition. Others look to automatic program generation techniques to convert high-level specifications into executable programs (Biggerstaff and Perlis, 1989). For example, a concrete component may be purchased in object-code form from a software parts supplier or it may be generated when needed by a program transformation tool. The 3C model is neutral with respect to this distinction.

In this chapter we are interested in studying the particular approach in which components are reused by *explicit programming* that composes them with other components. A component must be part of a larger program in order to be considered "reused." A stand-alone program that is simply executed over and over is not a reusable component by this definition.

What distinguishes the 3C model from the usual picture of software based on information hiding and abstraction principles? Obviously it is rooted in this philosophy, but it is more definitive in several important ways. The key features of the model from the standpoint of understanding reusable software components are the following:

- The abstract functional behavior of a piece of software is explicitly separated from the implementation of that behavior; i.e., concept is separated from content.

- For a particular abstract behavior there may be multiple implementations that differ in time/space performance or in price, but not in functionality; i.e., a given concept may have more than one content that realizes it.

- The external factors that contribute to the explanation of behavior are separated from that explanation; i.e., the context of concept is separated from the concept itself.

- The external factors that contribute to the implementation of behavior are separated from implementation code; i.e., the context of content is separated from the content itself.

Reuse can occur in several ways. Concepts are directly reusable abstract components. If there are multiple thick arrows into an abstract component then that concept is being reused within the program illustrated in the figure. A concept also can be reused in other programs, of course. Content can be reused indirectly, in that a concept can be reused and some of its many clients may choose the same concrete component to implement the concept. Finally, the various factors that make up the context of both concept and content (including mathematical theories and other abstract and concrete components) can be reused indirectly.

2.3 A Software Components Industry

If software is explicitly structured according to the 3C model, and reuse is practiced, what will a software components industry look like? All software activity will not fit into the scenario presented below, but there should be an economically significant fraction of all software design and development that will be based on this notion of reusable software components.

A mature software components industry will resemble the current electronic components industry. A client programmer will have one or more catalogs of standard reusable abstract components (concepts). As in electronics, these catalogs will be offered by different manufacturers or suppliers of reusable parts. Each abstract component in a catalog will have a formal description of its structural interface and functional behavior sufficient to explain what it does and how it might be incorporated into a client's system. It will also contain information about the performance and price of a particular concrete component (content) to implement it, which can be purchased from that supplier.

Many of the parts in a typical catalog will be standard in the sense that they will have the same structural interfaces and functional behaviors as corresponding parts in other manufacturers' catalogs. The rights to such standard concepts will be licensed from their original designers or will be in the public domain. The concrete components sold by different manufacturers generally will differ in implementation details and therefore in performance or price, however. A client will be able to substitute freely a concrete component provided by one supplier for a corresponding one provided by another, possibly with variations in function and/or performance available through parameterization (context).

This vision of a software components industry, on the surface so similar to McIlroy's of two decades ago, has many interesting implications. For one thing, it does not coincide with a possible world in which concrete components are assumed to be available in source code form. It seems that source code generally will not be sold under the above scenario. A reusable concrete component will be provided to clients in the form of object code (or some form not meant to be readable by humans).[1] No doubt this situation will seem distasteful to some, e.g. (Weiser, 1987), because source code availability does have some apparent advantages. For instance, when a program written for execution on a sequential machine is to be run on an architecture with vector processing units, it may be possible to parallelize some loops if source code is available. In-lining of procedures and optimization across procedure boundaries is another use for source code. Nonetheless, it is difficult to imagine a viable software components industry that is based on source-code reuse.

There are three main arguments for this conclusion. First, in the analogous electronic components industry, manufacturers do not publish or sell to clients the masks for their implementations of standard ICs, even though they literally give away catalogs containing interface specifications. Suppliers generally sell only sealed packages into which a client need not and should not look. We expect a software components industry to follow suit, assuming any technical barriers to this approach are conquered.

Second, providing source code to clients violates long-standing principles of information hiding and abstraction. It is a lot like explaining to a driver the gear-shift mechanism in his or her car by showing the blueprint for the transmission — except that revealing source code is more dangerous. Because software is so easily changeable, a user with access to source code seems to face an almost irresistible temptation to modify it. This leads to management difficulties, reliability problems, version control hassles, finger-pointing when technical support is requested, and so on.

Finally, identifying the fundamental differences between "source code" and "object code" is an important legal as well as technical issue. The legal dilemma facing a potential software components industry in this regard is serious. Internal data structures used by a component provider to represent various types and objects, as well as the algorithms to manipulate them, are not patentable under current law. Therefore, it has recently become common for companies to register copyrights for object-code versions of their programs and to maintain that the source code contains separately protected trade secrets (Tomijima, 1987). It is possible that such secrets could lose their protected status if provided to a client in a clearly revealed form, e.g., as source code (Samuelson,

[1] We use the term "object code" in the general sense of a program code that is not meant to be human-readable. Whether it is actually an intermediate form or executable code obtained as a result of compiler translation of source code is unimportant. An encrypted version of source code would also qualify under this definition, for instance.

1988). If this legal view prevails, there will be a significant economic risk for a manufacturer to sell source code for any but the most trivial components.

On the other hand, recent rulings have indicated that some courts consider the secrets in source code to be proprietary even if revealed to a client who paid for that source code (Samuelson, 1988). If this legal view prevails, there will be a significant economic risk for a client to buy source code because of the possibility of future litigation to prevent the independent use by the client of algorithms or data structures that are visible in purchased components.

During the period when neither legal position has clearly triumphed, source code sales are risky for both seller and buyer. Ultimately it will be hazardous for either or both. We therefore consider it unlikely that a mature software component industry will develop around source code reuse. The only realistic and economically viable situation is one in which abstract components are sold, licensed, or given away, and concrete components are sold by multiple suppliers in object-code form.

3. Defining a Reusable Concept

The years since McIlroy's proposal for a component-based software industry have brought significant advances toward realizing that vision. It is indeed surprising that we still seem so far from it, given the rate of technical progress. This may lead an uncritical observer to conclude that the major remaining impediments to widespread software reuse are non-technical ones. However, it should be apparent by the end of Sections 3, 4, and 5 that the technical front is not yet secure either.

We begin our analysis of design for reuse by considering how the behavior of an abstract reusable component can be defined.

3.1 Specification of Behavior

What could be worse than not reusing software? This question, posed by (Krone, 1988), has several possible answers. For one thing, an inappropriate component might be chosen — one whose actual behavior is misunderstood by the client programmer. This sort of mistake puts a damper on component reuse by reinforcing the "not invented here" syndrome. Furthermore, in a mature software component industry there will be several concrete components for any abstract component, giving a client the flexibility to choose price/performance characteristics that are compatible with the needs of the application. Interchangeable concrete components really *must* have the same functional behavior, so there can be no doubt about what that behavior is.

These observations suggest the following criteria for a behavioral specification and for a specification language, i.e., a language used to define reusable concepts:

- The specification of an abstract component must be clear, unambiguous, and understandable to a potential client and to a potential implementer.

- The specification of an abstract component must be free of implementation details in order to support a variety of concrete components that implement it.

We use the word *specification* to mean an explanation of both the structure and the behavior of a software component, i.e., its syntactic interface and its semantics. Occasionally, the same word is used elsewhere used to denote the syntactic interface alone. Ada uses it this way in the term "package specification."

The externally visible behavior of software is normally explained (if at all) by one of three techniques: comments embedded in code, informal metaphors, or formal mathematics. Source code comments have an obvious shortcoming with respect to the second criterion above. They also make no sense in the absence of the source code, a situation that will be inevitable in a mature software components industry. The other two methods are more reasonable. Both are abstract in the sense that they can be used to explain observable behavior in terms that factor out — and are neutral with respect to — the implementation details of any particular concrete component.

An example of a metaphorical description is “stacks are like piles of cafeteria trays.” This metaphor can be used to explain the visible effects of operations on stacks such as push and pop.[2] The general approach is to imagine a parallel in the physical world that acts as a *model* of an object in the computer. A metaphorical specification is the other side of the coin from the fundamental idea of object-oriented design (Cox, 1986; Meyer, 1988) in which a software object is considered to model a physical object. In either case it is important to establish a one-to-one mapping between physical and software objects, and this mapping can be interpreted in either direction. Just as a stack can be explained in terms of cafeteria trays, so might a program simulating cafeteria trays use a stack. A well-known use of a metaphorical explanation of software behavior is the Macintosh® desktop and toolbox. Metaphors are used not only to explain Macintosh applications to end-users, but to explain the behavior of many of the underlying reusable components upon which application programs are built (Apple, 1985).

A metaphorical description of software behavior is attractive because it is easy to read, but it is also inherently ambiguous because it is based on natural language (Liskov and Zilles, 1975; Meyer, 1985). By the first criterion above, it is therefore inappropriate as the sole specification of the behavior of an abstract reusable component. A natural language description may be part of an acceptable reusable component specification but it cannot do the job alone.

The remaining alternative is formal mathematical specification of abstract behavior. Here, a software object is modeled not by a real-world physical object but by an abstract mathematical object. This permits a specifier to write extremely precise statements about component behavior in a formal language such as predicate calculus, and it has the added advantage of making the specification amenable to machine processing.

It is imperative that a specification be comprehensible to a typical client programmer or implementer. Formal specifications do not have a reputation for being particularly understandable, but the “read-mostly” nature of reusable component specifications suggests some characteristics of a formal approach and a language to help mitigate this problem. The specification author may be assumed to be an expert in writing formal specifications, but the potential client or implementer may not be so well-versed in formal methods. Therefore, a specification approach that supports intuition, and a specification language that tends to be verbose rather than cryptic, seem desirable.

The first major technical question that must be addressed, then, is how to specify a concept in a way that has the two required properties of clarity and abstractness — which together imply formality (Wing, 1990). Two major approaches to formal specification have been developed: *algebraic*

[2] Using stacks to illustrate reuse ideas makes some critics uneasy because stacks seem too simple. Most of the fundamental ideas in this paper are illustrated with the stack example, even though they have been applied to the design of much more sophisticated abstractions. Stacks are ideal for pedagogical purposes and, as will be seen, are hardly as trivial as might be assumed. In fact, (Meyer, 1988) writes, “The stack example has been used over and over again to the point of becoming a cliché, but since this is because it is excellent, there is no reason to have any second thoughts about using it once more.”

® Macintosh is a registered trademark of Apple Computer, Inc.

specification (Guttag *et al.*, 1985; Wing, 1987) and *model-based* specification (Bjørner *et al.*, 1987; Spivey, 1989). Both are based on the idea that abstract component behavior can be explained through *mathematical modeling* of program objects by mathematical objects. The methods are virtually identical in spirit; the essential difference from the standpoint of specification is one of style. We use the traditional example of stacks to illustrate this distinction. Section 5.2 points out that there are other differences when it comes to using algebraic vs. model-based specifications for certifying that an implementation correctly meets its specification.

Fig. 2 shows a case where the abstract component being specified captures the idea of stacks. The intuitive behavior of stacks — the middle of the "Informal" column in Fig. 2 — is initially understood only informally by the designer, whose objective is to create a formal specification of it, as denoted by one of the gray rounded rectangles in Fig. 2.

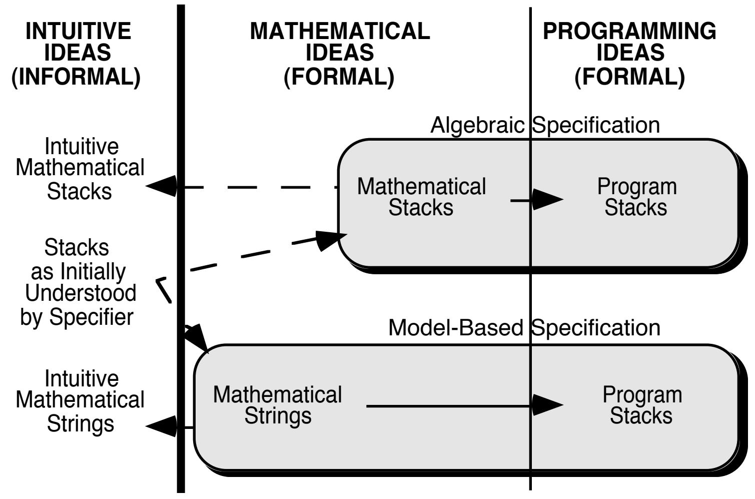
INTUITIVE IDEAS (INFORMAL)
MATHEMATICAL IDEAS (FORMAL)
PROGRAMMING IDEAS (FORMAL)
Algebraic Specification
Intuitive Mathematical Stacks
Mathematical Stacks
Program Stacks
Stacks as Initially Understood by Specifier
Model-Based Specification
Intuitive Mathematical Strings
Mathematical Strings
Program Stacks

FIG. 2. Algebraic and Model-Based Specification of Stacks.

Both an algebraic and a model-based specification have two pieces. The first is a formal description of mathematical theories that define some mathematical objects, and the second is a formal description of a program interface where the manner in which the mathematical objects model the program objects is explained. In Fig. 2 the two pieces of each specification are separated by the thin vertical line between mathematical and programming ideas.

3.2 Algebraic Specification

The top portion of Fig. 2 illustrates the nature of an algebraic specification of stacks. The idea is to define mathematical objects whose behavior is identical, or at least very close, to that of the corresponding program objects. The proximity of "Mathematical Stacks" to "Program Stacks" in Fig. 2 is meant to convey this close connection. An algebraic specification defines mathematical stack theory and then explains that program stacks act just like mathematical stacks. The modeling of program objects by mathematical ones is often defined implicitly through the identification of mathematical and program names of objects and functions. In fact, (Meyer, 1988) points out that not all authors even recognize the distinction between program objects and the mathematical objects that model them. Here we keep them distinct and given them different names.

An algebraic specification of stacks might start with the following definition of a theory of mathematical stacks. The text below is depicted by the block labelled "Mathematical Stacks" in Fig. 2. The language used is a compact dialect of RESOLVE, which includes constructs for formal specification using either the algebraic or model-based style.

```
theory STACK_THEORY_TEMPLATE (type T)
   type STACK
   functions
      NEW:      -> STACK
      PUSH:     STACK x T -> STACK
      POP:      STACK -> STACK
      TOP:      STACK -> T
   domain  conditions
      POP (s): not (s = NEW)
      TOP (s): not (s = NEW)
   axioms
      (1)       not (PUSH (s, x) = NEW)
      (2)       POP (PUSH (s, x)) = s
      (3)       TOP (PUSH (s, x)) = x
end STACK_THEORY_TEMPLATE
```

This part of the specification defines a schema for a family of mathematical theories, one for each type of item that can be in a stack. The generic theory — really a template for a family of parallel theories — is instantiated by providing a particular type for T. Mathematical type STACK is defined by axioms relating the mathematical functions NEW, PUSH, POP, and TOP, whose signatures are defined using standard notation for functions. NEW and PUSH are total functions, while POP and TOP are partial functions whose domains are characterized by the conditions stated. The axioms shown here are statements in predicate calculus (with equality, which is considered a built-in predicate for every mathematical type). Free variables in the axioms are assumed to be universally quantified. Here, the first axiom says the result of PUSH is never a NEW STACK; the second says the composition of PUSH and POP applied to a STACK gives the original STACK; and the third says any value of type T that is passed to PUSH is recovered if TOP is applied to the result of PUSH.

Writing the program interface portion of the specification is now a matter of declaring that program stacks behave like mathematical stacks. As noted above, this may be done by the default method that consists of using the same names for program types and operations as for the mathematical types and functions that model them and that define their behavior. Because NEW, PUSH, POP, and TOP are mathematical functions, though, implementation of this direct-definition interface in anything other than a functional programming language may prove troublesome. Consider a language such as Ada with procedures that can have side-effects on their arguments. In such a language the program interface specification must relate the mathematical functions to the behavior of a program module in which, for example, pushing and popping are done by procedures.

Here is how such a program interface might be explained in RESOLVE. The naming convention used here (not a language convention) is that mathematical names are in all upper-case, while program names have only the first characters of each word capitalized. This text is depicted by the block labelled "Program Stacks" in the algebraic specification portion of Fig. 2.

```
concept Stack_Template
   context
      parameters
         type Item
      mathematics
```

```
         theory STACK_THEORY is STACK_THEORY_TEMPLATE (math [Item])
      interface
         type Stack is modeled by STACK
         operation New (s : Stack)
            ensures    s = NEW
         operation Push (s : Stack, x : Item)
            ensures    s = PUSH (#s, #x) and x = #x
         operation Pop (s : Stack)
            requires   not (s = NEW)
            ensures    s = POP (#s)
         operation Top (s : Stack) returns x : Item
            requires   not (s = NEW)
            ensures    x = TOP (s)
   end Stack_Template
```

The concept definition begins with a description of its context. This consists of a type parameter Item that is supplied by a client to define Stacks of whatever kinds of values are needed. The concept is generic — it defines a family of related program concepts just as the theory template defines a family of mathematical theories. Type Item, however, is a program type. Like all program types it has a mathematical type that is used to model it, known as **math** [Item], which is used to instantiate STACK_THEORY_TEMPLATE. The particular theory of STACKS over this mathematical type (declared here to be named STACK_THEORY) is then used to explain the program interface.

The rest of the concept defines the syntax and semantics of the program interface to this abstract component. It states that program type Stack is modeled by mathematical type STACK. Program operations are defined through *preconditions* and *postconditions*, introduced respectively by the keywords **requires** and **ensures**. The meaning of this kind of specification is that, if the precondition is true when the operation is invoked, then the postcondition is true when it returns. Preconditions and postconditions are assertions in predicate calculus with equality in which the variables are the mathematical models of the operation's formal parameters.

In a requires clause, the values of the variables are those at the time of the call. The absence of a requires clause means that there is no precondition for calling the operation (equivalently, the precondition is "true"), i.e., the operation can be invoked under any circumstances. An ensures clause relates the values of the variables upon return to the values at the time of invocation and, in this way, explains the effect of the operation. In an ensures clause the value of a parameter at the time of the call is denoted by a mathematical variable with a "#" prefix before the parameter's name. For example, the assertion "s = PUSH (#s, #x)" means that the value of s upon return from the Push procedure equals the result of applying the mathematical function PUSH to the values of s and x that were passed to Push.

For a procedure operation (one without a **returns** clause) it is sometimes necessary to say that a parameter is unchanged by the operation. This is easily done by saying, e.g., "x = #x". For a function operation such as Top, the fact that the parameters are unchanged is implicit and need not be stated in the ensures clause. (In some languages, function operations are permitted to modify their arguments, but not in RESOLVE.)

3.3 Model-Based Specification

The bottom portion of Fig. 2 illustrates one possible model-based specification of stacks. Rather than defining a new mathematical theory of stacks, the specifier chooses from among a small collection of existing (and highly reusable) mathematical theories to define mathematical objects that can be used to model the desired program objects. The "semantic gap" between these familiar mathematical objects and the desired program objects is generally larger than it is for an algebraic specification; hence the larger distance from "Mathematical Strings" to "Program Stacks" in Fig. 2. The form of the specification is identical to that described above. The difference is in the choice of mathematical models.

Suppose we already have a formal definition of mathematical string theory in the same form as shown above for stack theory. This text is depicted by the block labelled "Mathematical Strings" in Fig. 2.

```
theory STRING_THEORY_TEMPLATE (type T)
   type STRING
   functions
      EMPTY:   -> STRING
      POST:    STRING x T -> STRING
   axioms
      (1)      not (POST (s, x) = EMPTY)
      (2)      POST (s1, x1) = POST (s2, x2) implies (s1 = s2 and x1 = x2)
end STRING_THEORY_TEMPLATE
```

The intended interpretation of these symbols is that a STRING is an ordinary (intuitive) string of items of type T, with EMPTY meaning the string containing no items, and POST (s, x) denoting the string obtained by appending the item x to the right end of string s.

The specifier in this case notes that a program stack can readily be modeled by a mathematical string where the top item of the stack is (arbitrarily chosen to be) the rightmost item of the corresponding string. This text is depicted by the block labelled "Program Stacks" in the model-based specification part of Fig. 2.

```
concept Stack_Template
   context
      parameters
         type Item
      mathematics
         theory STRING_THEORY is STRING_THEORY_TEMPLATE (math [Item])
   interface
      type Stack is modeled by STRING
      operation New (s : Stack)
         ensures   s = EMPTY
      operation Push (s : Stack, x : Item)
         ensures   s = POST (#s, #x) and x = #x
      operation Pop (s : Stack)
         requires  not (s = EMPTY)
         ensures   there exists x : math [Item], #s = POST (s, x)
      operation Top (s : Stack) returns x : Item
         requires  not (s = EMPTY)
         ensures   there exists s1 : STRING, s = POST (s1, x)
end Stack_Template
```

A concrete component with this interface should behave in exactly the same way as one with the interface specified in Section 3.2. That is, the two specification methods have been used here to define the same abstract behavior.

3.4 Language Issues

One of the notable missing pieces in three of the programming languages considered here (Ada, C++, and Eiffel) is a sub-language for writing formal specifications. C++ has no constructs whatsoever for this purpose. Eiffel permits "assertions" in a program, but they are intended primarily to support debugging through run-time monitoring of the state of object representations. Assertions can describe certain properties of program behavior but are not powerful enough to support abstract specification in general. The limitation can be seen by observing that Eiffel's assertions are simple boolean expressions involving program functions and procedures applied to concrete representations of program objects. They are not statements in predicate calculus involving quantifiers or mathematical functions applied to abstract models of program objects. Published classes from the Eiffel library are not specified in any formal language, although the code contains some Eiffel assertions.

While Ada itself has no constructs for formal specification, an Ada extension in the form of an annotation language called ANNA (Luckham *et al.*, 1987) has been defined for this purpose. Assertions in ANNA generally are comparable to those in Eiffel and (although it was created with the objective of permitting formal specification and verification of Ada programs) ANNA has been used primarily for run-time monitoring. Such assertions generally cannot be used to specify abstract behavior for the reason noted above. ANNA also permits a designer to write "axioms" about user-defined types, but it is not clear whether these are also intended to define directly the behavior of program functions and procedures or the behavior of their mathematical models. Specification with ANNA axioms syntactically resembles algebraic specification, but the semantic underpinnings are not clear from the few published examples we have seen.

Some specification languages are not tied directly to an implementation language. Two modern specification languages of this kind support both the algebraic and the model-based style: Larch (Guttag *et al.*, 1985; Liskov and Guttag, 1986; Wing, 1987) and Z (Spivey, 1989). Both permit the specifier to define mathematical theories and then to define abstract program interfaces using those theories.

In Larch, a mathematical theory is written in the Larch Shared Language (Guttag *et al.*, 1985; Guttag and Horning, 1986a) and a program interface specification is written in an implementation-language-specific Larch Interface Language (Wing, 1987). Larch can be used to write specifications in either the algebraic or the model-based style, but the main catalog of published examples (Guttag and Horning, 1986b) clearly favors the algebraic style. That is, a program concept typically is explained by a mathematical model that is devised especially to explain it. There is a library of reusable mathematical properties that reflect classical ideas, including binary relations, partial orders, and groups. There are also less-reusable theories of stacks, queues, containers, priority queues, and binary trees. The latter are used to define their program counterparts, while the former are used primarily to explain constraints on the parameters to generic concepts.

By contrast, published examples in Z such as those in (Spivey, 1989; London and Milsted, 1989) emphasize model-based specification. Z has been used to define a relatively small but powerful library of mathematical theories for, e.g., sets and mappings. Each program object is explained in terms of some combination of these mathematical objects. A specifier generally need not dream up a new mathematical theory for each new concept, but instead identifies an appropriate mathematical model for it from among alternatives in the library.

RESOLVE is similar to Larch and Z in that it has separate constructs for defining mathematical theories and program interfaces. However, the implementation language is assumed to be RESOLVE. This means it is possible to define a fixed set of proof rules that relate abstract components to concrete components, thereby supporting verification of correctness. This connection, which is missing in Larch and Z, is explained briefly in Section 5.2.

The main advantages of algebraic specification over model-based specification are the flexibility to define new mathematics when necessary, and the inherently close connection between program objects and their mathematical counterparts. In a strictly model-based approach the explanation of program objects uses only existing mathematical theories. If a new program concept is exotic enough, then this constraint might require the specifier to use a mathematical model that does not define what might directly explain the program operations. This may result in requires and ensures clauses that are long and convoluted and, consequently, difficult to write and to understand. The program interface part of an algebraic specification should always be comparatively straightforward. This difference is illustrated schematically in Fig. 2, where the distance between “Mathematical Stacks” and “Program Stacks” in the algebraic specification is less than the distance between “Mathematical Strings” and “Program Stacks” in the model-based specification. Examination of the two versions of the Stack_Template concept reveals only a slight advantage to the algebraic approach in this case. Our experience shows this is typical.

The major advantages of model-based specification over algebraic specification result directly from the reuse of mathematical theories. For one thing, the model-based specifier does not need to know how to define mathematical theories — a non-trivial task at best. In fact, much of the early work on algebraic specification concentrated on methods for assisting the designer in demonstrating logical properties of the custom-made theories used in algebraic specification (e.g., soundness, consistency, relative completeness). Defining a new theory remains a difficult chore. An ill-advised definition can lead to serious trouble, especially when proving properties about programs. Fig. 2 also illustrates this other side of the coin. Because the specifier presumably already knows, understands, and trusts the previously-defined theory of “Mathematical Strings,” it is a smaller step to understand that theory than to define and show the required properties of the new theory of “Mathematical Stacks.”

Perhaps most important from the standpoint of reuse, a model-based specification is probably easier for a prospective client to understand than an algebraic specification. We know of no controlled experiments to support or refute this claim, but it is entirely plausible. Clients, like specifiers, can be expected to learn to interpret (and trust) a relatively small, fixed class of theories. Most of the relevant mathematical ideas — integers, sets, functions — are already known to programmers as a result of the normal educational process. Other important ones can easily be learned. It is asking far more to expect a client to decipher a new mathematical theory in order to understand each new abstract program component. Fig. 2 suggests this difference with the length of left-pointing dashed arrows in the “client understanding” direction: from formalism to intuition.

The total distance between “Program Stacks” and the client’s intuition is shown as about the same for both approaches to specification because the client’s understanding of a specification includes both an understanding of the mathematical theories *and* an understanding of the connection between the mathematical and program objects. That the two methods are about equal in this regard is probably a fair characterization overall in the sense that some people feel more comfortable with algebraic specification, others with model-based specification. Our experience teaching both undergraduate students and practicing software engineers in industry suggests that a model-based approach is generally preferable. This is the approach used in the remainder of the chapter.

4. Designing a Reusable Concept

Having a notation to express abstract component designs is a big step toward promoting reuse. Of course it is still necessary to design good reusable components, which has proved to be a surprisingly difficult quest. Some reasons for this and some proposed guidelines to assist a designer of reusable concepts are explored in this section.

4.1 Efficiency and the Relationship Between Concept and Content

What else could be worse than not reusing software? An apparently reusable abstract component may be designed poorly from the standpoint of reuse. There are many ways in which a design may be inadequate but still tolerable, but there is one problem that history has shown is sure to frustrate reuse: inefficiency (SofTech, 1985). An abstract component with only inefficient concrete components to implement it tempts a client to "roll his/her own." A poorly designed abstract component may even inherently *rule out* efficient realizations.

Is it even possible to have reusable components that are both flexible and efficient? Reusable software has a reputation for being inefficient — one of the problems that tempts a source code owner to modify it. This perception is based partly on folklore about an intrinsic trade-off between generality and efficiency. There is no theoretical reason for believing in such a trade-off. It is not surprising that it is observed in current practice, though, because a typical reusable software component is based directly on the designs, data structures, and algorithms found in standard computer science textbooks: classical stacks, lists, symbol tables, sorting algorithms, and so on. Difficulties arising from their potential reusability were not at issue in the original designs of these structures, and their performance as the basis for reusable components suffers as a result.

Even in cases where an abstract design is a good one there may be no efficient implementation currently on the market. This is almost as serious a problem as the impossibility of an efficient implementation, because many of the advantages of reuse are not achieved if a client reuses only a concept or some other aspect of a "high-level design" and has to build a new implementation for it. The client programmer's productivity will surely suffer, not only during coding but more importantly during maintenance (Edwards, 1990). It has been estimated that 70% of the cost of software is attributable to maintenance (Boehm, 1987). Concrete components purchased from outside suppliers generally will be certified to be correct implementations of their abstract counterparts and normally will not be available in source-code form. Therefore, a purchased component usually will require — in fact, will permit — *no* maintenance effort by the client. A custom part produced in-house is subject to the usual problems and expenses of maintenance. Moreover, a concrete component that forces the programmer to go through difficult gymnastics in order to achieve the desired efficiency is likely to be even more of a maintenance headache than the average program.

Methods for designing concepts and languages for defining content of components, then, are subject to an important criterion:

- An abstract component design, in conjunction with the programming language in which it is to be realized, must not make it difficult or impossible to build multiple efficient concrete components that implement the design.

An implementation of a concrete component with a given abstract interface, such as the Stack_Template of Section 3, can sometimes be created automatically from the formal behavioral specification. This kind of implementation, also called an *executable specification*, is exemplified by the language OBJ (Goguen, 1984). An OBJ specification of an abstract component can be thought of as a set of equations that define a mathematical theory, or it can be treated as an executable program by interpreting the defining equations as rewrite rules. The performance of a concrete component that is constructed in this way may or may not be acceptable. We do not further consider this method of implementation because one of the basic features of a reusable software components industry will surely be a potential client's freedom to choose among various concrete components that realize the same abstract component. Some of these choices might be automatically generated from specifications and prove useful as "rapid prototypes," but at least for the foreseeable future their performance will easily be dominated by cleverly-devised programs written by humans. The manual approach is therefore the focus here.

In this view an implementer of a concept invents concrete representations for program objects and algorithms to manipulate them. For example, there are several possible representations for Stacks with different algorithms for the program operations New, Push, Pop, and Top. More sophisticated abstract components such as those supporting associative searching, sorting and selection, and graph manipulation have more interesting performance trade-offs, of course. Their alternative implementations may differ in execution-time performance for the various operations or along any other dimension except abstract functional behavior. Some concrete components may dominate others completely, but in general it is far more likely that competing ones are simply incomparable in the sense that one is "better" for some operations or along some dimensions, while others are "better" for other operations or along other dimensions. A client program may use any concrete component that correctly implements the abstract interface. Regardless of the implementation, the client always reasons about program variables as if they are abstract mathematical variables, e.g., he/she reasons about Stacks as though they are STRINGs.

The need for efficiency has two major implications. It calls for a set of design guidelines to help concept definers create abstract components that admit efficient concrete components. It also makes it important to identify programming language constructs that support efficient implementation of reusable components, as well as those that thwart it. These issues are discussed in the remainder of Section 4.

4.2 General Design Guidelines for Reusable Abstract Components

As noted by (Bentley, 1982), it has long been a goal of software engineers to make their field more like the other engineering disciplines by developing design guidelines and standards. The art involved in software design will always remain important, but experience that leads to good design must be captured and transmitted to others. A recent report (Computer Science and Technology Board, 1990) calls for an effort to produce "handbooks" of software engineering knowledge similar to those used by other engineers. Here we concentrate on design-for-reuse guidelines that might be included in such a handbook, especially guidelines related to efficient implementation.

When dealing with efficiency we usually mean execution time in the "big-O" sense. Space is important, too, but typically it is not the deciding factor in whether a potentially reusable component is reused in practice. The constant additive or even small constant-factor overhead associated with making a procedure call or dereferencing a pointer, for example, is also of little concern when compared to order-of-magnitude penalties imposed by inappropriate data structures and algorithms (Gannon and Zelkowitz, 1987; Muralidharan, 1989).

There is of course a considerable body of well-known work on efficient data structures and algorithms, much of which has made its way into the early computer science curriculum through textbooks such as (Martin, 1986; Feldman, 1988). But there has been little published work on efficiency considerations that arise directly from reusability. These features are often subtle but very important. Some recent works in which these issues are considered, such as (SofTech, 1985; Harms, 1990; Sitaraman, 1990), have not yet been published in the archival literature. Some of the most important efficiency-related questions are therefore considered below in detail.

Although the discussion is not couched in terms of reuse, (Liskov and Guttag, 1986) write about three important properties of an abstract specification: clarity, restrictiveness, and generality. "Clarity" means that a specification is understandable, and "restrictiveness" means that it states everything the designer wishes to state. Both are important attributes of a good design but neither has much to do with whether efficient implementation is possible. "Generality" means that a specification is sufficiently abstract and not too restrictive to permit a variety of acceptable and efficient implementations. In the setting of a software components industry in which multiple concrete components are expected for each abstract component, this is certainly a worthwhile design objective.

Using what (Liskov and Guttag, 1986) call a "definitional" style of specification, as opposed to an "operational" one, is a major step toward generality. A formal specification in either the algebraic or the model-based style must be abstract by its very nature because it explains program behavior through mathematical modeling. It is still possible to define a model that is highly suggestive of a particular implementation. For example, a Stack might be modeled as an ordered pair: an integer and a mapping from integer to **math** [Item]. This abstract model suggests that a Stack might be represented as a record comprising an integer index of the top Item and an array of Items in the Stack. Other representations are still possible; the question is whether a typical programmer is likely to think of anything else after seeing this sort of operational hint in the specification.

Unfortunately, even with a completely abstract and definitional formal specification method it is amazingly easy to define behavior that rules out efficient implementations of the underlying concept. In fact, because of the functional style of an algebraic specification, it is easier to define behavior in that style that cannot be implemented efficiently than to define behavior that can be. The classical Stack_Template of Section 3 is an excellent example of this phenomenon, as we note in Section 4.4.

Other positive qualities of reusable component designs are proposed by (Booch, 1987). He defines a "primitive" operation for a type as one that cannot be implemented efficiently without access to the underlying representation of the type. To avoid possible confusion with the idea of a built-in type or operation (which also are often described as "primitive"), we prefer to call such an operation a *primary* operation. All other operations are called *secondary* operations. For example, the Push operation of the Stack_Template is primary because there is no way to obtain its effect using a combination of the other operations. An operation to reverse a Stack is secondary because it can be implemented by calls to Push, Pop, etc.

Primary operations usually should be primitive in the sense that they make only incremental changes to their parameters' abstract values. The effect of an operation that makes large changes often can be obtained by code that comprises more primitive operations. This observation can be used as a check on the quality of a component design. Moreover, notice that an operation is primary *relative* to other exported operations. Different subsets of all possible operations for a type might be considered primary. It is up to the designer of an abstract component to choose an appropriate set of primary operations.

A primary operation is so fundamental that it must be implemented together with the code that defines the representation of some type. Secondary operations can be implemented by layering on top of the primary operations, or they can be implemented with direct access to the underlying representation of some type. The distinction of primary vs. secondary is defined in terms of what operations *can be* implemented by layering, not in terms of which ones actually *are* implemented by layering. Among the many advantages of layering is that when a different realization of a basic abstract component is substituted, there is in principle no need to re-code, re-certify, or even re-compile layered secondary operations. However, in some situations it is possible to achieve significant efficiency gains by implementing certain secondary operations directly, just like primary operations. Language mechanisms that support the distinction between primary and secondary operations, and between layered and direct implementations of the latter, are discussed in Section 5.1.

Booch's notions of "sufficiency" and "completeness" are closely related ideas. A component providing a type is sufficient if it also exports enough operations to characterize the type, and it is complete if it exports all operations deemed by the designer to be useful. A component is practically worthless if it does not satisfy the sufficiency criterion, but a reusable component need not be complete in the sense that all (or even any) secondary operations must be exported by the basic abstract component. It should be *potentially* complete in that it should be possible to build any interesting secondary operation by layering on top of the primary operations. Most potential manipulations involving any type do not need access to its underlying representation; this is the entire idea behind data abstraction. Secondary operations generally should *not* be exported as part of a basic reusable abstract component but should be added as extensions or enhancements.

In considering the question of objectively evaluating the quality of Ada packages, (Embley and Woodfield, 1988; Gautier and Wallis, 1990) apply two well-known characteristics of software in general: "coupling" and "cohesion." A component should be self-contained (low coupling) and not further decomposable (high cohesion). An abstract specification approach and language mechanisms such as generics for dealing with conceptual context make it easier to achieve these goals. It is still up to the designer, however, to create a good component design. An abstract component should exhibit low coupling, i.e., its behavioral specification should not depend on other abstract components — and certainly not on any concrete components. By the same token a component should exhibit high cohesion, i.e., it should encapsulate a single idea. It should not contain a jumble of almost-independent ideas in an end-run attempt to satisfy the coupling criterion.

Notice that the coupling rule is intended primarily to guide the design of the most basic reusable components. A typical basic component such as the Stack_Template should export a type and its primary operations. Extensions of such a component are possible, though. For example, an abstract component that enhances the functionality of the Stack_Template with an operation to reverse a Stack obviously should be explained in terms related to the behavior of Stacks. This much coupling is pemissible. However, the explanation of Stacks themselves should not rely on the behavior of Arrays just because some implementation of the Stack_Template might use them.

In summary, the literature discusses a number of general properties of good reusable component designs. They are meant to be interpreted as general guidelines for design or as properties of good designs, not as hard-and-fast laws of reuse. We can rephrase them as follows.

A reusable abstract component should:

- Be clear and understandable ("clarity").
- State everything about the behavior that is expected of a correct implementation — and nothing more ("restrictiveness").
- Support a variety of implementations, and especially not rule out efficient ones ("generality").
- Export operations whose functionality is so basic it cannot be obtained by combinations of other exported operations, i.e., it should export primary operations ("primitiveness").
- Export primary operations that together offer enough functionality to permit a client to perform a wide class of interesting computations with the component ("sufficiency").
- Not export operations that can be implemented using the primary operations, i.e., secondary operations, unless the component is an extension or enhancement of a more

basic reusable component that exports the primary operations (“potential completeness”).

- Not depend on the behavior of another abstract component for explanation of its functionality, unless it is an extension or enhancement of that component (“low coupling”).

- Encapsulate a single concept that cannot be further decomposed, e.g., a single type (“high cohesion”).

4.3 Corollaries

There seems to be considerable agreement among reusable component designers that the above general guidelines are reasonable. This agreement may be due in part to the lack of precision with which they are often stated. A number of more specific consequences can be deduced from these guidelines and, judging from other design-for-reuse proposals and from actual published designs of reusable components, some of the more obvious corollaries are quite controversial at the detail level. Some of these are illustrated below using the Stack_Template as a simple example. A more detailed critique and redesign of the Stack_Template is presented in Section 4.4.

4.3.1 Initialization and Finalization

The ability to prove program correctness formally is tantamount to the ability to do careful informal reasoning about client program behavior. It is vitally important, not just an esoteric desire for formality (Shaw, 1983). One important implication is that every program variable must be considered to have an initial abstract value. For example, every Stack must have an initial abstract value in order to permit verification of the correctness of Stack_Template’s clients. The reason is simply that following declaration of a Stack and prior to its abstract initialization it makes no sense to consider the Stack to be modeled by a mathematical STRING. In particular, if there is no initialization of a Stack at the abstract level it may be impossible to interpret the requires or ensures clause of the first operation involving that Stack.

Initialization at the abstract level requires initialization at the concrete level, at least for some potential implementations of an abstract component. The argument here is that every implementation of the Stack_Template, for example, must set up some concrete representation of each Stack in order to “prime” the sequence of operations that Stack will participate in during its lifetime. This initial concrete configuration represents the initial abstract value. Except in rare cases where any random setting of bits in memory can be interpreted to represent a value of the mathematical model — none of which is obvious for Stacks — *some* code must be invoked to construct a legitimate initial configuration.

Although the need for initialization of variables is well-known and widely acccepted, its impact on reusable component design is not universally acknowledged. There is no general agreement on how to accomplish initialization, either. For example, an early Ada style manual (Hibbard *et al.*, 1983) recommends that each user-defined type should have an explicit initialization procedure. A later compendium of reusable component design guidelines for Ada programmers (SofTech, 1985) also calls for the ability to do “creation” and “termination” of objects. But it provides the following suggestion for how to achieve this: “Implement all private types as records so that automatic initialization may be guaranteed.” A similar guideline appears in (Gautier and Wallis, 1990), but the detailed recommendation (also for Ada) is still more specific: The concrete representation of each variable should be set to a value that is recognizable as “uninitialized,” and every operation should check for uninitialized arguments and invoke the code to initialize them as necessary. (Hibbard *et al.*, 1983) argue convincingly that this method is needlessly inefficient.

Any recommendation for implementing the general rule that variables should be initialized must work uniformly for *every* conceivable concrete component that implements a concept. Otherwise client code might have to be changed if one concrete component is substituted for another with ostensibly the same functionality. The generality criterion suggests that there is only one rational way to approach initialization that respects the underlying objective of efficiency: There should be an explicit operation to initialize a Stack, and it should be called by a client immediately after declaration of a Stack variable.

Similarly, when a variable is no longer needed, there should be a way to reclaim the memory occupied by its concrete representation. Otherwise the client may eventually exhaust available storage. Some implementations might rely on garbage collection for this. Other implementations, however, are more efficient if they can explicitly deallocate the memory used by their representations. The generality criterion again suggests the only rational approach: There should be an operation to finalize a Stack, and it must be called by a client after a Stack is no longer needed. In a concrete component that relies on garbage collection the finalization operation may do nothing, but the hook should be there for an alternative implementation that explicitly manages its own storage.

In general we therefore recommend:

- A reusable abstract component that exports a type should export an initialization operation and a finalization operation to be invoked at the very beginning and at the very end, respectively, of the lifetime of any variable or object of that type ("initialization/finalization corollary").

In Ada and Eiffel the client programmer is responsible for invoking initialization and finalization operations explicitly. For reusable component designs in these languages we suggest the following conventions for initialization and finalization operations. A concrete component implementing an abstract component that exports a type should be required to provide code for these operations. A client should be expected to invoke the initialization operation for every variable immediately after entering the scope of its declaration, and to invoke the finalization operation immediately before exiting that scope (Muralidharan and Weide, 1990).

C++ supports compiler-generated calls to operations for this purpose ("constructors" and "destructors") but does not require that a component have them. We suggest that a constructor and destructor be defined for every reusable component that defines a new type.

RESOLVE enforces automatic initialization and finalization. Every type must have these two operations, which are invoked by compiler-generated calls (at entry and exit of a variable's scope). Explicit client calls to them are not needed and are not permitted.

4.3.2 Defensive vs. Non-Defensive Designs

A property of a component called "totality" is suggested by (Wing, 1987) as a desirable quality. Roughly stated, an abstract component has the totality property if each procedure or function has a well-defined result for all values of its input arguments — in short, if it does not have a requires clause. This attribute is easy to check, and methods for achieving totality are readily discovered if a design is deemed wanting in this respect. Another view of totality is as "defensive programming," which is offered by (Berard, 1987) as a necessary property of reusable software. The designs proposed by (Hibbard *et al.*, 1983; Liskov and Guttag, 1986; Booch, 1987) are also generally defensive, i.e., “error-catching.” General rules supporting this approach are proposed in (SofTech, 1985; Gautier and Wallis, 1990).

Despite the above claims, it is far from clear that reusable components should be defensive. If the client of a piece of software is an end-user who may make a mistake with it, then that software clearly should be defensive: It should catch user errors and do something predictable and friendly under all circumstances. However, reusable software components are not invoked like application programs. They are embedded in other software that uses their services. There is no question that an operation should not be called when its arguments do not satisfy the requires clause; the question is whether the client program or the reusable component should perform the check. It is important to adopt a consistent approach to this issue because it is obviously not a good idea for neither side to worry about it, but it is impractical for both sides to do so. Redundant checking is inefficient, leading (Liskov and Guttag, 1986) to recommend in-line expansion of procedure calls and subsequent source-level optimization to overcome it. But designs based on redundant checking bring what (Meyer, 1988) calls “conceptual pollution” to the overall system. A convention about who is responsible for checking requires clauses should be adopted. Like Meyer we suggest that basic reusable components’ operations should *not* catch violations of their requires clauses. However, it should always be possible for a client to use the exported operations to check any requires clause and, if desired, to build a defensive version of a concept as a layer on top of the more basic non-defensive one.

We therefore recommend another corollary that can be viewed as a natural consequence of the generality, primitiveness, and potential completeness criteria:

- A basic reusable abstract component should not export defensive operations, but should permit a client to define a corresponding defensive component in which the operations can be implemented by layering on top of the more basic component (“non-defensiveness corollary”).

A component designed according to this guideline is not responsible for catching or for handling errors that might occur while it is executing, and therefore should not (among other things) raise any exception conditions resulting from violations of its requires clauses. As noted above, however, component designs in languages with exception-handling constructs (e.g., Ada and Eiffel but not C++ and RESOLVE) often are designed defensively. It seems tempting to use exception-handling when it is available. The reusability guidelines for Ada in (SofTech, 1985) contain a specific suggestion to this effect: “For each assumption a subroutine depends on to operate correctly [i.e., requires clause], define an exception that is to be raised when the assumption is violated.” The next guideline offered is, “For every situation that would raise an exception, define a function that indicates whether the exception would be raised.” Similar suggestions are offered by (Gautier and Wallis, 1990).

Notice that if one follows the second suggestion in the design of a basic reusable component, then it is easy to build a component that follows the first suggestion without loss of efficiency — by layering on top of the original component — but not vice versa. That is, if there is an operation to check the requires clause of every operation then a defensive design for an operation can raise exceptions as necessary. The following schema illustrates this:

```
if check of requires clause finds that it is satisfied
   then call original operation
   else raise exception
end  if
```

On the other hand, if the most primitive operation available is designed to raise an exception when its requires clause is violated, then a non-defensive version built on top of it must also pay for the code that performs the check. This observation means that a defensive design is never the most primitive one.

The question of whether to raise an exception when a requires clause is violated is interesting in its own right. Raising an exception during component debugging seems reasonable (Luckham *et al.*, 1987; Meyer, 1988). For delivered components this strategy is less attractive, though. Exceptions are a dangerous method of altering program flow of control when an ordinary if-statement or while-loop can do the job. Again (SofTech, 1985) suggests a curious approach: "Instead of requiring the user to find out if a stack is empty prior to calling the Pop operation, design the [reusable software component] so that Stack.Pop raises the exception Stack.Empty or assigns True to the flag Empty if the stack is empty." Presumably the normal way for a client to process all Items in a Stack is to write the following kind of code:

```
    loop
       ...
       Pop (s);
       ...
    end  loop;
    ...
 exception
    ...
    when Stack.Empty ...
```

We cannot recommend a reusable component design approach that suggests that a client program should include such convoluted code for such a simple job. If exceptions are to be raised at all, their use should not demand that a client use exception handlers in order to do quite ordinary and expected things.

4.3.3 Copying and Equality Testing

Demanding sufficiency, potential completeness, low coupling, and high cohesion still leaves considerable flexibility for the designer because there may be many different subsets of all operations that can serve as the primary operations. Intuitively, the designer's objective is to choose a "basis" for all computations with the type — a set of operations that "span" the set of computations that a client might wish to perform, yet are "orthogonal" in the sense that none is implementable using a combination of the others.

An important question that arises here is how to characterize the set of computations that a client may wish to perform. One property of a truly reusable component is that not all possible uses of it are known to the designer. However, this does not prevent the designer from anticipating what operations these uses might involve. For instance, one test of functional completeness of a design is part of the non-defensiveness corollary: It should be possible to code tests of all requires clauses of the exported operations. If there is no primary operation to test a particular requires clause it should be possible to write a secondary operation that does so. Building a defensive version of the component as a layer on top of the more primitive non-defensive one is a possible use that a designer *should* anticipate.

Two other things that can be anticipated as possible client needs are copying and equality testing for an exported type. In the case of a composite type such as a Stack, which holds Items of another arbitrary type, these operations cannot be implemented without copying and equality testing operations for the type Item. Their abstract effects can be specified, however, in reusable form:

```
 concept Copying_Capability
```

```
    context
      parameters
        type Item
    interface
      operation Replica (original : Item) returns copy : Item
        ensures   copy = original
  end Copying_Capability

  concept Equality_Testing_Capability
    context
      parameters
        type Item
    interface
      operation Are_Equal (x : Item, y : Item) returns control
        ensures   Are_Equal iff (x = y)
  end Equality_Testing_Capability
```

These concepts can be generic because RESOLVE's specification language (like most) includes equality as a predicate for every mathematical type. No particular theories are needed to explain either operation. Any implementation, of course, will be type-specific. By the way, the specification of Are_Equal introduces an operation that returns **control**. RESOLVE has no built-in types — even Boolean — so what would in other languages be boolean-valued functions are called *control operations* that return "yes" or "no." Control operations are invoked to compute conditions to be tested in if-statements and while-loops in client programs and, like function operations in RESOLVE that return results, are not permitted to modify their parameters.

The generality criterion implies that copying and equality-testing should not be treated as built-in operations. It is possible in some languages (e.g., Ada) to use default assignment and equality-testing operators even with a private type, i.e., one whose representation is hidden from the client. As shorthand for "y := Replica (x)" one may write "y := x", and for "Are_Equal (x, y)" one may write "x = y". However, the shorthand versions operate on concrete representations, not abstract values, and therefore may not act as Replica and Are_Equal are specified to behave. Assignment may result in unintended aliasing and subsequent incorrect program behavior that copying with Replica cannot produce; see Section 4.4. The built-in "=" operator may return the wrong result if the mapping between concrete representations and their mathematical models is not a bijection (Martin, 1986); e.g., two Stacks may test unequal because their representations are not identical, even though as abstract mathematical STRINGs they are equal.

Since relying on language primitives to do copying and equality-testing is dangerous, we recommend that a designer anticipate at least two more possible client needs:

- A reusable abstract component should export primary operations that are adequate to permit a client to code Replica and Are_Equal as secondary operations or, if necessary, the component should export them as primary operations ("copying/equality-testing corollary").

In our experience this is a very powerful test of the functional completeness of a design and leads to interesting components, especially for more sophisticated applications such as associative searching (Sitaraman, 1990) and graph manipulation.

Copying and equality testing are also interesting from the language standpoint. As noted above, Ada permits these built-in operators to be used with private types. A component designer can prevent their use by declaring a type *limited* private, however, and we recommend this approach (Muralidharan and Weide, 1990). Other authors, e.g. (Booch, 1987), also struggle with this issue.

Booch disagrees with our conclusion, declaring an exported type like Stack limited private but noting that if a generic type parameter like Item is limited private, then the implementation of the component may not use the built-in assignment or equality-testing operators on it. From the standpoint of reasoning about correctness this is precisely the reason a type parameter *should* be limited private. There might be a slight efficiency penalty for writing "y := Replica (x)" in place of "y := x" in those instances where the built-in operators would work correctly for the actual type involved, but there is no question about the semantic effect of the statement in the generic implementation. Booch's approach in which type parameters are declared private, not limited private, also limits severely the composability of components. For instance, a client cannot define Stacks of Stacks of any type.

Eiffel permits the assignment operator to be used with any type, but its meaning is different for built-in types than for user-defined types. Assignment copies abstract values for built-in types but copies references (i.e., pointers to objects) for user-defined types; a similar distinction is made for the equality-testing operator. The non-uniform semantics necessitated by this approach is disturbing. It also makes it impossible to reason about Eiffel programs without introducing a level of indirection in the abstract mathematical models of all user-defined types, significantly complicating formal specifications and their use in verification (Shaw, 1981).

C++ permits the built-in assignment and equality-testing operators to be used, with semantics similar to Eiffel's. The component designer may (but is not obligated to) override the default effects by providing special code for copying and equality testing of abstract values. Again, we suggest this always be done in C++ in order to keep reasoning at an abstract level where possible.

RESOLVE does not have the usual built-in assignment or equality-testing operators, so the effects of Replica and Are_Equal are obtained by invoking the operations explicitly. There is a function assignment statement — "y := Replica (x)" is an example — but the right-hand side must be a function invocation. It may not be a variable. There is no implicit copying in RESOLVE.

4.4 An Example: Critique and Redesign of the Stack_Template

The Stack_Template of Section 3 embodies the classical design of a stack component that appears — although usually without a formal specification — in a variety of modern texts, presumably as an example of a good abstract design. It has its formal basis in the earliest works on algebraic specification (Liskov and Zilles, 1975) and no doubt has been used in hundreds of programs over many years. How does it stack up against the criteria outlined above? Sadly, despite its historical importance and its prominent place as probably the first potentially reusable component seen by a computer science student, the traditional design of the Stack_Template is not a good one from the standpoint of reuse. There are several reasons for this conclusion.

4.4.1 Initialization and Finalization

The Stack_Template design does not satisfy the initialization/finalization corollary. Initialization is presumably the purpose of the New operation. Nothing in the formal Stack_Template specification says this, however. Some authors, e.g., (Jones, 1988), try to augment the specification with an additional requires clause for each operation to the effect that "s is valid." This also becomes part of the ensures clause of New. The requires clause of New becomes "s is not valid." This approach is somewhat confusing but probably acceptable in an informal specification, but what does it mean in a formal specification? What does "validity" of a Stack s mean in the mathematical STRING model? There is no analogous idea on the mathematical side: A mathematical variable *has* a value of its type, even if that value is unknown. Reasoning about program variables as though they have the values of their mathematical models is therefore compromised by the extra-specificational notion of "validity" of a concrete representation.

The Stack_Template also lacks an operation to finalize a Stack, thereby ruling out any implementation that can achieve efficiency advantages by avoiding garbage collection. Indeed there are such representations of Stacks (Pittel, 1990).

Fixing these problems is easy. From now on we assume that the specification of a type includes an assertion about the initial value of every variable of that type, as is the case in RESOLVE. A prototypical variable of a type, which is used in the initial value assertion, is called an *exemplar*. For the Stack_Template we augment the type definition and remove the New operation, leaving us with:

```
concept Stack_Template
   context
      parameters
         type Item
      mathematics
         theory STRING_THEORY is STRING_THEORY_TEMPLATE (math [Item])
   interface
      type Stack is modeled by STRING
         exemplar  s
         initially s = EMPTY
      operation Push (s : Stack, x : Item)
         ensures   s = POST (#s, #x) and x = #x
      operation Pop (s : Stack)
         requires  not (s = EMPTY)
         ensures   there exists x : math [Item], #s = POST (s, x)
      operation Top (s : Stack) returns x : Item
         requires  not (s = EMPTY)
         ensures   there exists s1 : STRING, s = POST (s1, x)
end Stack_Template
```

There is no need to specify the effect of finalization because it is invoked only after a Stack is no longer needed and, therefore, has no particular effect on the abstract model of a Stack. The finalization operation is usually important only as a hook for a concrete component to manage its own dynamically allocated memory.

4.4.2 Potential Completeness and Non-Defensiveness

The revised Stack_Template above requires a Stack to be non-empty before a client may invoke Pop or Top on it. This design therefore is not defensive. However, it is not potentially complete. The component offers no way for a client to check this requires clause. The non-defensiveness corollary suggests that it is advisable to redesign the Stack_Template with either an emptiness test or an equality test (but not both) as a primary operation.

With an equality-testing operation it is possible to check whether a Stack is EMPTY by comparing it to a newly declared (initially EMPTY) Stack. Alternatively, with an operation to test emptiness and the other Stack_Template operations — plus an equality-testing operation for Items — it is possible to check whether two Stacks are equal. The primitiveness criterion suggests the simpler operation is more appropriate as a primary operation. Therefore, we add a control operation called Is_Empty, leaving us with the following design:

```
concept Stack_Template
```

```
    context
       parameters
          type Item
       mathematics
          theory STRING_THEORY is STRING_THEORY_TEMPLATE (math [Item])
    interface
       type Stack is modeled by STRING
          exemplar  s
          initially s = EMPTY
       operation Push (s : Stack, x : Item)
          ensures   s = POST (#s, #x) and x = #x
       operation Pop (s : Stack)
          requires  not (s = EMPTY)
          ensures   there exists x : math [Item], #s = POST (s, x)
       operation Top (s : Stack) returns x : Item
          requires  not (s = EMPTY)
          ensures   there exists s1 : STRING, s = POST (s1, x)
       operation Is_Empty (s : Stack) returns control
          ensures   Is_Empty iff (s = EMPTY)
 end Stack_Template
```

4.4.3 Choice of Primary Operations

Minimality is the principal objective in choosing a primary set from among all conceivable operations, i.e., the set of primary operations usually should have minimum cardinality while still satisfying the properties of primitiveness, sufficiency, and potential completeness. To select among many possible such sets of operations of the same cardinality, though, a designer should consider efficiency: Which choices of primary operations lead to potentially efficient implementations of secondary operations, and which thwart efficiency? Sometimes it is possible to identify a set of primary operations that dominates all others in this respect. The Stack_Template is an example of this. However, the operations Push, Pop, and Top in the current design do not constitute a well-chosen set of primary operations.

The first evidence for this conclusion is that Push and Top are potentially quite inefficient. Notice that there is no reason for any restriction on type Item and in fact none is specified. A client instantiating Stack_Template may replace Item by any program type, including simple types like Integer or more complex types like Queue of Integers or even Stack of Integers. The problem with the Push operation is that it demands that x not be changed, but also demands that the (old) value of x become the top Item of s. The implementation of Push therefore must place a *copy* of x onto Stack s. Because type Item may end up being one whose representation is large and expensive to copy, the Push operation may run very slowly. Consider its execution time when x is a Queue of Integers, for example.

A similar problem with the design stems from the semantics of the Top operation. Again, because Item may be any type, copying the top Item of s to return to the caller may be expensive. This situation is acceptable if Item is restricted to simple types such as Integer. But if there are no restrictions on type Item then the inherent copying designed into the Push and Top operations is problematical.

Both efficiency problems noted above can be traced to what might be called the “copying style” of design and programming. This style is taught to most programmers and is encouraged by Ada , C++, Eiffel, and their cousins, as well as by functional programming languages. How are stacks designed and implemented in these languages? Published interfaces typically mimic the original

Stack_Template concept of Section 3 (or are very similar). Published implementations invariably use assignment statements and other more subtle ways to make copies of Items. This is no surprise — they are forced to do so by the design of the abstract interface.

Consider an Ada generic package body to implement the Stack_Template. Suppose Stack is declared as a private type with a typical representation — a record with two fields: contents (an array of Items) and top_index (the index in the contents array of the top of the Stack). Ignore the fact that the array has a fixed maximum size while the Stack it represents can be arbitrarily large; the same phenomenon is observed with the more complex code of a linked representation such as that proposed in (Booch, 1987). The code for Push, Pop, and Top might look like this:

```
package body Stack_Template is
   ...
   procedure Push (s : in out Stack; x : in Item) is
   begin
      s.top_index := s.top_index + 1;
      s.contents (s.top_index) := x;         -- copying an Item
   end Push;
   procedure Pop (s : in out Stack) is
   begin
      s.top_index := s.top_index - 1;
   end Pop;
   procedure Top (s : in Stack) return Item is
   begin
      return (s.contents (s.top_index));    -- copying an Item
   end Top;
   ...
end Stack_Template;
```

The comments mark where inefficiency might arise. If the data structure representing the abstract value of an Item is an array or a record with many fields, for example, the two commented statements are expensive in execution time despite their deceptive simplicity.

Possibly recognizing this, some authors, e.g., (Stubbs and Webre, 1987), replace the original Pop and Top operations with a combined operation — also called Pop but with different behavior — producing yet another variation on the design:

```
concept Stack_Template
   context
      parameters
         type Item
      mathematics
         theory STRING_THEORY is STRING_THEORY_TEMPLATE (math [Item])
   interface
      type Stack is modeled by STRING
         exemplar  s
         initially s = EMPTY
      operation Push (s : Stack, x : Item)
         ensures    s = POST (#s, #x) and x = #x
      operation Pop (s : Stack, x : Item)
         requires   not (s = EMPTY)
         ensures    #s = POST (s, x)
      operation Is_Empty (s : Stack) returns control
         ensures    Is_Empty iff (s = EMPTY)
   end Stack_Template
```

The advantage of this approach is that the implementation of the new Pop need not copy an Item. It simply removes the top value from s and returns it to the caller in x. If a client program doesn't need a copy of this Item, then it doesn't have to pay for making one, as it would if it called Top.

The new design is also more consistent with the general reuse guidelines. For one thing, it has fewer operations defined in the interface and is therefore more concise and probably more understandable. Furthermore, it is easy to implement the original Pop (call it Pop_And_Discard) and Top as secondary operations using the new Stack_Template design. Here is RESOLVE code for them:

```
      operation Pop_And_Discard (s : Stack)
         local  variables
            x: Item
      begin
         Pop (s, x)
      end Pop_And_Discard

      operation Top (s : Stack) returns x : Item
      begin
         Pop (s, x)
         Push (s, x)
      end Top
```

This implementation of Pop_And_Discard may be slower than if it is coded as a primary operation, but only by a small constant factor due to an extra layer of procedure call. Top is slower by a constant factor for the same reason. On the other hand, if the new Pop is implemented as a secondary operation using Top and Pop_And_Discard, then the client pays for copying an Item, like it or not.

How does the above code for Top copy an Item? It is done by the Push operation, which is defined in such a way that its name should be Push_A_Copy. This observation suggests yet another improvement to the Stack_Template design that replaces the original operation with a new Push that "consumes" the Item being pushed onto the Stack. The ensures clause of Push now says nothing about the value of x upon return from the operation, so the implementer is free to return any value for that parameter. It is no longer necessary to copy an Item in order to implement Push. This change leaves us with the following (final) redesign of the Stack_Template:

```
   concept Stack_Template
      context
         parameters
            type Item
         mathematics
            theory STRING_THEORY is STRING_THEORY_TEMPLATE (math [Item])
      interface
         type Stack is modeled by STRING
            exemplar  s
            initially s = EMPTY
         operation Push (s : Stack, x : Item)
            ensures   s = POST (#s, #x)
         operation Pop (s : Stack, x : Item)
            requires  not (s = EMPTY)
            ensures   #s = POST (s, x)
         operation Is_Empty (s : Stack) returns control
            ensures   Is_Empty iff (s = EMPTY)
     end Stack_Template
```

An interesting and obvious question is whether the redefined behavior of the Push operation is really what a client program might want. There are grounds for believing that in most uses of Stacks it is exactly what is needed. There is usually no reason to keep a copy of an Item that is pushed onto a Stack. After all, the entire idea of using a Stack is usually to keep track of information that will be needed later in LIFO order. A client normally does not need to have a separate copy of that information at the same time.

Again, however, if the behavior of the original Push operation (call it Push_A_Copy) is really needed, it is easy to layer it on top of the new Push operation without incurring a significant performance penalty, while the converse is not true. Here is RESOLVE code for Push_A_Copy:

```
    operation Push_A_Copy (s : Stack, x : Item)
       local variables
          y: Item
    begin
       y := Replica (x)
       Push (s, y)
    end Push_A_Copy
```

The Stack_Template now exports only primary operations. Secondary operations — those identified so far are Push_A_Copy, Pop_And_Discard, and Top — should be defined by a separate abstract component or by an enhancement or extension of this one; see Section 5.1.

4.4.4 Swapping

A very careful reader may wonder exactly *how* the new Push and Pop operations can be implemented without copying an Item. In Ada, for example, it seems that something like the original code still must be used, including assignment statements involving Items:

```
    procedure Push (s : in out Stack; x : in out Item) is
    begin
       s.top_index := s.top_index + 1;
       s.contents (s.top_index) := x;         -- copying an Item
    end Push;
    procedure Pop (s : in out Stack; x : in out Item) is
    begin
       x := s.contents (s.top_index);         -- copying an Item
       s.top_index := s.top_index - 1;
    end Pop;
```

The understandable but erroneous conclusion that copying Items is still necessary even with the new abstract design is another result of our long training and experience with the copying style of programming. To see that there is an alternative, suppose the assignment statement of Ada is replaced or augmented with a *swap* statement of the form "x :=: y" where x and y are variables of the same type whose values are exchanged by the statement.[3] The precise meaning of swapping can be defined by imagining it is a call to a procedure with the following specification:

[3] There seems to be no reason a swap statement could not be added to Ada, C++, Eiffel, or almost any similar language with virtually no impact on the rest of the language.

```
operation "_:=:_" (x : Item, y : Item)
   ensures    x = #y and y = #x
```

Push and Pop can now be coded as follows, with swap statements replacing the assignment statements involving Items:

```
procedure Push (s : in out Stack; x : in out Item) is
begin
   s.top_index := s.top_index + 1;
   s.contents (s.top_index) :=: x;        -- swapping Items
end Push;
procedure Pop (s : in out Stack; x : in out Item) is
begin
   x :=: s.contents (s.top_index);        -- swapping Items
   s.top_index := s.top_index - 1;
end Pop;
```

What is the advantage of swapping over copying? It seems at first glance as if swapping two Items must be as expensive in execution as *three* assignment statements. For Items whose representations are large and complex, swapping seems three times *less* efficient than copying.

However, notice that the desired semantic effect of swapping two abstract values can be achieved in implementation simply by swapping pointers to the data structures that represent those values; see Fig. 3. This means that if a compiler (without the knowledge of the programmer) adds an extra level of indirection to all variables whose representations are larger than a pointer, then the swap statement can always be compiled into code that takes exactly three pointer move instructions in a typical instruction set. Swapping two Items therefore can be implemented to run in a (very small) constant amount of time, regardless of how large or complicated the representations of those Items might be. Moreover, the same code that swaps two Items also swaps two Stacks or any other pair of values of the same type.

The universal efficiency of swapping has interesting implications. First, it is better to design operations whose implementations can swap values rather than copying when those values might be of arbitrary types. For example, an operation to access a value in an array can be designed so that the array is modified in the process, by swapping the value in the indexed position with one of the arguments to the procedure. The usual fetch and store operations are secondary operations with this design. Using swapping rather than assignment as a built-in operation leads to a slightly different programming style than for other Pascal-like languages. It is rather easy to learn, though, because it is a minor variation, and it usually results in more efficient programs than can be produced by using previously published designs for the components involved. We have designed dozens of concepts using the swapping style and have found that a couple of new programming idioms are all that one needs to learn in order to feel comfortable with it (Harms, 1990).

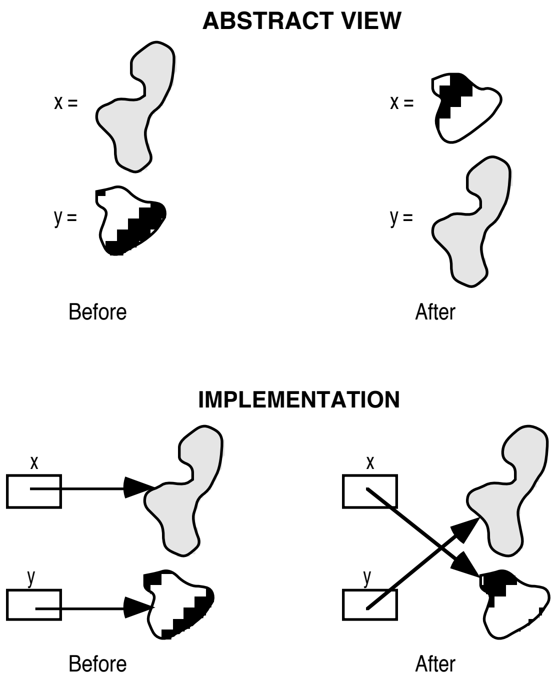

FIG. 3. Abstract View and Implementation of Swapping.

4.5 Chasing Efficiency at the Expense of Correctness

Before leaving the question of implementation guidelines we consider another obvious, popular, and (in some circles) explicitly recommended approach to removing the inefficiency of copying large Item representations: Represent a type as a pointer to the data structure that represents the abstract value (SofTech, 1985; Booch, 1987). Most programming languages encourage this. The terminology of Ada even gives the impression that a client programmer does not have to see the "private" part of a package specification because the choice of a type's representation does not matter from the standpoint of functionality. This is not true. In fact the inefficiency of the Ada code implementing the original Push and Top operations in Section 4.4 is the *good* news about it. The bad news is that it may be incorrect. It mistakenly assumes the assignment statement "y := x" always has the same effect as "y := Replica (x)".

Here is the code in question:

```
package body Stack_Template is
   ...
   procedure Push (s : in out Stack; x : in Item) is
   begin
      s.top_index := s.top_index + 1;
      s.contents (s.top_index) := x;         -- copying an Item (?)
   end Push;
   ...
   procedure Top (s : in Stack) return Item is
   begin
      return (s.contents (s.top_index));    -- copying an Item (?)
   end Top;
```

```
      ...
   end Stack_Template;
```

To see the problem, consider Stack_Template instantiated as follows in a client program:

```
   package Stack_Of_Ints is new Stack_Template (Integer);
```

This package instantiation replaces Item in generic package Stack_Template with the built-in type Integer, to define Stacks of Integers (type "Stack_Of_Ints.Stack"). Now consider the following variable definitions:

```
   i1, i2:     Integer;
   si1, si2:   Stack_Of_Ints.Stack;
```

The abstract effect of the assignment "i1 := i2" is of course that, afterward, i1 has the same abstract value as i2, and i2's value is not changed. The implementation of the assignment statement is that a copy of the representation of the abstract value of i2 is made and becomes the representation of the abstract value of i1. A subsequent change to i1 by, e.g., "i1 := i1 + 53" has absolutely no effect on the abstract value of i2. This matches our intuition about what an assignment statement does. We expect "si1 := si2" to have a similar abstract effect. The abstract value of si1 (a mathematical STRING) becomes equal to the abstract value of si2, which does not change. A subsequent change to si1 by, e.g., "Push (si1, i1)" has no effect on the abstract value of si2 just as it has no effect on the abstract value of i2 — even if it follows the statement "i2 := i1".

All this is true if the representation of type Stack is the one first hypothesized for the private part of Stack_Template, i.e., a record with two fields. It is not true, however, if the representation of type Stack is a pointer to such a record. Then the assignment of a Stack variable copies only this pointer, not the record to which it points. Of course this is efficient compared to copying the record but it leads to incorrect abstract behavior. Following "si1 := si2" the statement "Push (si1, i1)" results in the abstract values of both si1 and si2 being changed.

The effect of copying only a pointer to the representation data structure, not the data structure itself, is known as *aliasing* or *structural sharing*. It is a situation in which a data structure is known by two or more different names. If the effect of aliasing that occurs inside a component is visible in the client program — if it manifests itself as a "linkage" between two variables that is not explained in the abstract mathematical model used to reason about the behavior of those variables — then it is easy to write programs that appear to work but are really incorrect. Code walk-throughs and testing can easily fail to identify aliasing errors because the implicit linkage between variables may arise only in certain circumstances, only for certain types used in generic package instantiations, or only in certain combinations or sequences of statements. The aliasing problem is not unique to Ada, of course, but arises in every language that has pointer types (Meyer, 1988). In fact, (Hoare, 1983) remarks that because of aliasing, "introduction [of pointers] into high-level languages has been a step backward from which we may never recover."

Notice that swapping does not introduce aliasing even though representations involve compiler-introduced pointers in order to permit efficient implementation of the swap operator. In RESOLVE the language rules completely prevent implicit aliasing (which in most other languages can also occur as the result of constructs other than the assignment statement, e.g., parameter passing). Like all other types, pointers are not built-in but are exported by formally specified concepts in order to permit careful reasoning about program behavior. The undeniably advantageous efficiency effects of aliasing that can be achieved by disciplined and judicious use of pointers in other languages are obtained in RESOLVE by encapsulating clever pointer tricks into reusable abstract components (Pittel, 1990).

5. Other Issues

While space limitations prevent detailed treatment of other important issues related to design and implementation of reusable software components, we briefly discuss two more significant problems that affect reuse. Both demand additional attention before they can be considered to be solved.

5.1 Defining Context

What else could be worse than not reusing software? A not-quite-suitable component might be available — one with no hope of adaptation to the specific needs of the client. On the surface the main problem here may seem to be the frustration experienced by the client. However, such predicaments over time will lead to a software components industry with a proliferation of concepts that differ only in minor ways. There will be an even greater proliferation of concrete components. This can only lead to confusion about which components do what, to questions about why they are considered different, and to a net reduction in reuse compared to a more structured situation. A client in this world may spend an inordinate amount of time trying to locate and understand available reusable components only to end up starting over from scratch.

Relatively minor differences among similar abstract components and among similar concrete components are considered in the 3C model as part of the context of concept and content, respectively. Stated in these terms, the success of a software component industry will be based partly on the following requirements:

- The approach and language for specifying an abstract component must include mechanisms to permit concept adaptation (i.e., behavioral adaptation) by the client through the use of conceptual context.

- The approach and language for implementing a concrete component must include mechanisms to permit content adaptation (i.e., performance adaptation) by the client through the use of implementation context.

One of the problems with a proliferation of components is the increased difficulty of searching through them to find a particular behavior of interest. There has been considerable work in this area recently. Some researchers classify components using standard methods from library science, while others draw on techniques from artificial intelligence (Prieto-Diaz, 1987). The underlying motivation for such efforts is the belief that a useful catalog of reusable components will be so large that it will be difficult for a typical client programmer to find anything in it without sophisticated computer assistance.

The need (as opposed to the opportunity) for computer-assisted searching among reusable components is dubious. Indexing reusable concepts on the basis of abstract functionality within application domain, and then organizing the variety of concrete components that implement each concept on the basis of performance and similar attributes, results in a natural hierarchy that should keep a typical client's search space quite manageable. Furthermore, a large factor reduction in the size of that space can be achieved if there are effective mechanisms for parameterizing the context of both concept and content (Edwards, 1990). For example, the "Booch components" in Ada (Booch, 1987) include over 20 variations and implementations of essentially a single concept: stack.

Both the ease of locating reusable abstract components and client understanding of the ones that are found are influenced by an ability to factor context from concept. Similar benefits are available on the concrete component side. For example, suppose a client program needs both stacks of integers and stacks of characters. If a single flexible concrete component implements these two obviously related concepts then the client wins on two counts. First, he/she only has to purchase a single adaptable concrete component rather than two more specialized concrete components. Second, the total volume of object code in the client's product is smaller if these two variations of stacks share code.

As suggested above, there are really two distinct kinds of context: fixed and parameterized. Both kinds of conceptual context are illustrated in the Stack_Template. An example of fixed conceptual context is the declaration of the theory of mathematical STRINGs to explain Stacks in the Stack_Template. A client has no choice in the theory that is used in the specification. On the other hand, the client is permitted to choose the kinds of Items that will be stored in a Stack; this is parameterized conceptual context. A similar distinction exists on the implementation side, where a typical concrete component relies on some fixed implementation context (e.g., context brought in using the Ada **with** clause) as well as on client-supplied generic parameters.

Design for reuse implies that both functionality and performance should be as adaptable as possible. Technically, this means that the designer of a basic reusable component should strive to make context parameterized rather than fixed. Marketing concerns in the software components industry may result in purchased components that are easy-to-understand and/or easy-to-instantiate specializations of underlying reusable concepts and implementations that are highly parameterized (Musser and Stepanov, 1989). This kind of adaptation will be done by the supplier, not the purchaser, but it will still rely on methods for factoring context from concept and content, and on mechanisms for parameterizing context.

The two most important language mechanisms that have been developed for this purpose are genericity and inheritance (Meyer, 1986). Both have been widely adopted in practice through their incorporation into practical programming languages, but in their current forms they remain disturbingly unsatisfactory in principle.

5.1.1 Genericity

The Stack_Template example used throughout this chapter is *generic*, i.e., it is parameterized by the type Item. In effect the Stack_Template is a schema, or pattern, or template — hence its name — for a family of reusable abstract components. A client is responsible for creating an *instance* of the schema by substituting an actual type for the formal type parameter Item. The mathematical theories for STACKs and STRINGs introduced in Section 3 are generic in the same sense, although the term is ordinarily used to describe program concepts rather than mathematical theories.

The limits of genericity as a mechanism for parameterizing context are not well understood (Sitaraman, 1990). This is partly because the extent to which a language supports genericity has a tremendous influence on the power of the idea. C++ does not have genericity and Eiffel uses it only to parameterize types, as we have seen here. Ada and RESOLVE extend genericity in important ways beyond type parameterization and give the flavor of the potential power of the mechanism (Musser and Stepanov, 1989).

Reusable Ada components are most easily designed as packages. Each package has a header, called a "package specification," that defines the syntax of its interface. It has a separately compilable "package body" that defines the implementation. A parameterized package is called a "generic package." Parameterization by a type is only one of the ways a component can be generic in Ada. Values and program operations may also be used as generic parameters. For example, using the style of design described in Section 4, a designer might write the following generic package specification in Ada:

```
generic
```

```
      type Item is limited private;
      with procedure Initialize (x : in out Item);
      with procedure Finalize (x : in out Item);
      with procedure Swap (x, y : in out Item);
   package Stack_Template is
      ...
   end Stack_Template;
```

One problem with Ada generics is that no semantic information is provided to restrict the actuals that may be substituted for formal generic parameters (Livintchouk and Matsumoto, 1984). It is possible for a client to instantiate a package with any procedures that have the calling signatures of the corresponding formals. For instance, the initialization procedure for type Item can be passed legally for both Initialize and Finalize in the example above because they have the same structural interface. This problem is attributable to Ada's lack of an associated specification language. However, it suggests that an effective mechanism for parameterizing context should include not only program parameters but mathematical parameters that can be used to explain the behavior of the program parameters.

RESOLVE contains integrated specification and implementation languages in which both mathematical and program ideas may be passed as generic parameters. Formal generic program parameters must include explanations of their expected behavior, and this is often expressed using other (mathematical) generic parameters. An ordinary compiler cannot check that an actual parameter has the proper behavior, but proof rules of the language permit a verifier to do so in order to guarantee that a program is not certified as correct if it contains a component that is improperly instantiated (Krone, 1988).

Another interesting aspect of Ada generics is that there is no distinction between conceptual and implementation context. The type Item above is required to *explain* Stacks, but procedures Initialize, Finalize, and Swap are needed only to *implement* Stacks. If any conceivable implementation of a concept needs an operation involving the types of other generic parameters, then that procedure also must be listed as a generic parameter in the package specification.

While the seriousness of this problem is not evident from the Stack_Template example, consider the "abstract" design for a reusable set concept reported by (London and Milsted, 1989). Although it is written in Z and has no connection whatsoever to a particular implementation language such as Ada, the specification includes a hash function in the abstract component interface. All the set operations are explained by their impacts on a hash table, even though this is only one possible representation for sets. The authors note that the Smalltalk sets upon which their design is based are implemented using hashing, but do not apologize for the violation of abstraction or its inhibiting effect on reusability. In fact, they state that the objective of the Z specification is to "model essentially all the details of an industrial-strength implementation of sets." In this case the inability to separate conceptual from implementation context seems to have contributed to the unnecessary mixing of concept and content.

To see that a hashing function need not participate in this abstract component's specification, consider a similar concept called Partial_Map_Template by (Sitaraman, 1990). This abstract component captures the idea of associative searching by modeling a search as the evaluation of a partial function from type Domain to type Range. These types are the only conceptual parameters. As with sets, one possible implementation of the Partial_Map_Template is to use hashing, another is to use a binary search tree, and there are many others. The first implementation needs an operation to compute a hash function given a value of type Domain, while the second needs an operation to compare two Domain values with respect to some total order. In Ada, both of these operations must be additional parameters listed in the generic package specification even though neither is necessary to explain the abstract behavior of the Partial_Map_Template. Otherwise the concept cannot support these two functionally indistinguishable concrete components that might implement it. If other possible representations might involve other operations on Domain or Range values they also must be added to the concept's generic parameter list. An alternative in Ada is to have separate package specifications for each implementation, but this suggests that there are multiple concepts when in fact there are multiple implementations for a single concept.

In RESOLVE there are separate syntactic slots for conceptual context (as seen in Section 3) and for implementation context. A client of Partial_Map_Template who instantiates the hashing implementation provides a hash function; one who instantiates the binary search tree implementation provides a comparison function. The jury is still out on how a component designer or supplier should use this mechanism to trade off between complexity and flexibility of client parameterization, but it seems clear that separate parameterization of concept and content is important for reuse in the 3C model and in the components industry scenario (Sitaraman, 1990).

Compared to most languages Ada has a rather comprehensive mechanism for generics, but it does not allow a package to be a generic parameter. However, RESOLVE does permit instances of concepts to be parameters to other concepts and to implementations. This is necessitated by the need to perform strong compile-time type-checking, and it facilitates composition of components that otherwise would have to be designed in violation of the low-coupling guideline (Harms, 1990). Other interesting frontiers of genericity are explored by (Sitaraman, 1990).

5.1.2 Inheritance

Inheritance is widely considered another promising approach to factoring context from concept and content. Inheritance is a way of defining a new component as a variation on an existing one (in the case of *multiple* inheritance, two or more existing ones). A hierarchy of components is defined by identifying, for each new component, the existing components to which it is related through inheritance. A new component is usually an extension of its *parent(s)* in the hierarchy in that it provides the same services and possibly more. There are two important relationships between components in a system based on inheritance. One is the usual client-component relationship in which the client uses a component by invoking its services. The other is the inheritance relationship in which a component (the *heir*) inherits from its ancestors.

Language specifics are again very important influences on the manner in which one thinks about and uses inheritance. C++ has single inheritance, whereas Eiffel offers multiple inheritance. Ada does not provide an inheritance mechanism. RESOLVE offers a limited form of inheritance called enhancement. The Eiffel view of inheritance is the basis for most of the discussion that follows.

Inheritance may be used in several ways (Meyer, 1988). First, in Eiffel as in most languages with inheritance, there is no explicit separation of concept from content. However, it is possible for a component's operations to be "deferred"; C++ has a similar notion called "virtual." A deferred operation has no code to implement it. It is a placeholder for the name of an operation that an heir must define in order for execution to be possible. For example, in the case of the Stack_Template concept we might define a class in which all the operations are deferred. One heir of this class might represent Stacks using arrays, another might use lists. The parent class stands for the concept while the two heirs provide alternative ways of implementing its content, as shown in Fig. 4. This use of inheritance can be considered a way to separate concept from content and to relate them in the obvious way.

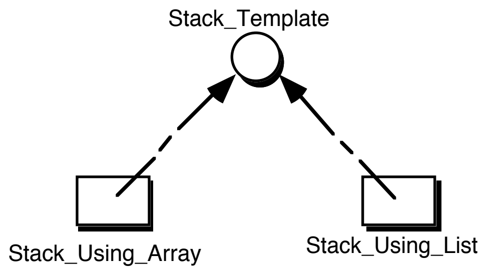

FIG. 4. Using Inheritance to Separate Concept from Content.

Another important use of inheritance is to change (presumably slightly) the functionality and/or implementation of an existing component. An heir is permitted to see the internal representational details of its ancestors' data and operations. It may extend an ancestor by adding new data representation features, by adding code for new operations, and/or by overriding (in effect replacing) the existing code for some or all of the ancestors' operations. However, an ancestor may not be changed directly in order that its clients remain unaffected by the addition of heirs. This use of inheritance may be considered to have two distinct purposes: to define a new concept by describing its relation to existing abstract components (which then constitute its conceptual context), or to define new content by taking advantage of its relation to existing concrete components (its implementation context). The first purpose is sometimes called *type* or *specification inheritance* and the second *implementation* or *code inheritance*.

Inheritance therefore does triple-duty: It is used to separate concept from content, concept from conceptual context, and content from implementation context. Its role as a single language mechanism that supports three important objectives is part of the appeal of inheritance. However, there is reason to suspect it may not do particularly well at any of them precisely because it must be adequate for all three. For instance, it has been noticed that the latter two purposes tend to conflict (LaLonde, 1989). An inheritance hierarchy that effectively separates concept from conceptual context seems only vaguely related to a hierarchy that effectively separates content from implementation context. There could be two different inheritance hierarchies, one for specification inheritance and one for code inheritance, but we do not know of a language in which the details of such a split have been worked out.

There is even some doubt about whether code inheritance should be permitted at all. A serious problem with it is that a concrete component's implementer must understand the implementation details and subtle representational conventions of all of its ancestors in order to implement that component correctly. Unless care is taken it is possible to introduce components that seem to work properly yet, by manipulating their ancestors' internal data representations, violate subtle and implicit conventions that the ancestors require for correctness. Information hiding is thus compromised (Snyder, 1986; Muralidharan and Weide, 1990). Moreover, source code for ancestors must be available in order for new components to inherit code from them. This situation is not likely to be viable in a mature software components industry. In fact, some authors have recently concluded that code inheritance should be abandoned altogether because it actually discourages design for reuse (Raj, 1990) and because its advantages largely can be obtained — without the disadvantages — through effective use of parameterization (Raj and Levy, 1989; Muralidharan and Weide, 1990; Sitaraman, 1990).

Few similar objections have been raised to specification inheritance, which continues to seem attractive as a way of relating a new concept to existing ones. The challenge is to achieve this objective without introducing a high degree of coupling among concepts comparable to the high degree of coupling among implementations that arises from code inheritance. With specification inheritance a client must understand the behaviors of all of a component's ancestors in order to understand that component. One possible solution to this problem is to try to keep the inheritance hierarchy looking like a forest of fairly short and independent smaller hierarchies. This characterization does not describe the inheritance hierarchies of component libraries currently in use in inheritance-based systems, however.

Recognizing the difficulties with code inheritance and the importance of concept hierarchies that are short and independent of each other, RESOLVE includes two different mechanisms that have the same benefits as a restricted form of specification inheritance. The first is generic parameters that are instances of concepts. Using this kind of genericity it is possible, for example, to specify an operation that can be used to convert, say, a Stack from any representation to any other. The implementation of this conversion operation can be layered on top of the Stack_Template so it works no matter what underlying representations are involved. Other interesting uses of genericity are explained by (Harms, 1990; Sitaraman, 1990).

A new concept also may be defined as an *enhancement* of another. For example, if an operation to reverse a stack is needed, the final Stack_Template of Section 4.4 may be augmented with a new concept:

```
concept Reverse_Extension enhances Stack_Template
   interface
      operation Reverse (s : Stack)
         ensures    s = REVERSE (#s)
end Reverse_Extension
```

The conceptual context of Reverse_Extension is all of Stack_Template, including the mathematical machinery declared there. Assuming String_Theory includes the definition of REVERSE (although it is not shown in Section 3), the specification of the Reverse operation for Stacks is easy. Reverse may be implemented as a secondary operation by layering on top of the Stack_Template operations. It is also possible to build a new concrete component for the combined concept "Stack_Template with Reverse_Extension" by implementing the original Stack_Template operations plus Reverse in such a way that all operations have access to the concrete representation of Stacks. This can result in more efficient execution. For example, Reverse can be implemented to run in constant time rather than linear time even as the other operations suffer only a small constant-factor performance degradation. Any other composite concept may be created by mixing and matching the possibly many enhancements of a basic reusable concept. The composite concept may be implemented by layering using secondary code for the enhancements, or by reimplementing the entire composite concept.

While this approach does not offer all of the flexibility of inheritance, it avoids the most serious disadvantages and offers a few advantages of its own. It is too early to tell whether the trade-off is a good one. More examples and more research are needed to clarify the many issues involved in separating context from concept and content. Some particular problems of interest include investigation of the interactions among the three uses of inheritance, the extent to which they conflict from the standpoint of reuse, and the general difficulty of managing the details of highly parameterized context of sophisticated reusable components.

5.2 Certifying That Content Correctly Implements Concept

What else could be worse than not reusing software? An incorrect concrete component might be chosen — one that is not a correct realization of the corresponding abstraction. Clients of reusable software components, like those of electronic components, will expect the parts they purchase to work correctly. An electronic component, even after exhaustive testing of the logical design, might fail because of physical defects introduced in the manufacturing process. Presumably failures of software components will be attributable mostly to logical errors in implementation or coding. This suggests that an auxiliary industry of component certifiers might arise. Perhaps manufacturers, clients, and/or standards bodies will develop an independent “Underwriters’ Laboratory” for software components.

A viable software component industry therefore will demand that:

- A concrete component must be certifiably trustworthy, i.e., it must correctly implement the corresponding abstract component.

At first glance it appears there are two alternative approaches to certifying that a particular content faithfully implements the corresponding concept. *Verification* (also called *proof of correctness*) is a formal demonstration that the code implements what the formal specification demands. There are two parts to such a proof: showing that the implementation is correct if a call to an operation returns, and showing that it does in fact terminate. *Testing* is done by running an operation on test data and inspecting the results to see that it actually does what is intended. Of course, this is only possible when the operation does return.

The terms verification and testing as used here should not be confused with the terms “verification and validation” that have been used to describe similar but not identical processes by, e.g., (Wallace and Fujii, 1989). Like many technical words, these terms have slightly different meanings in different sub-communities of software engineering.

The view that both verification and testing are suitable for certifying that content correctly implements concept does not tell the whole story. Fig. 5 illustrates the informal and formal aspects of a typical abstract component A and one concrete component C that implements it. Both A and C are written in a formal language with well-defined syntax and semantics. Section 3 explains why this must be so for A, and most computer scientists are already comfortable with the idea that an implementation language can be treated formally. The figure also shows the natural language description R — perhaps a metaphorical explanation as discussed in Section 3 — of the intuitively required abstract behavior which is (supposedly) defined formally in A. Finally, it shows the observed execution behavior E of the implementation defined formally in C.

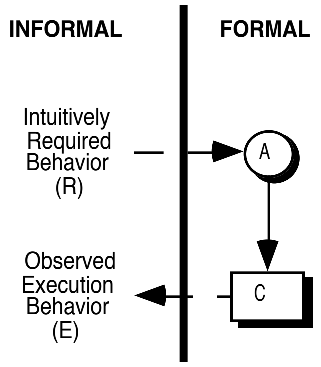
INFORMAL
FORMAL
Intuitively Required Behavior (R)
A
Observed Execution Behavior (E)
C

FIG. 5. Informal and Formal Aspects of a Software Component.

A central issue in certification of correctness is to demonstrate that the “is correctly implemented by” relation (denoted by the arrow from A to C) holds. With the appropriate mathematical machinery this is a formally-defined property relating two formally-defined entities. If it can be proved that the relation holds, this fact is demonstrable by a formal process of symbol manipulation following well-defined rules of logic. Such a proof is mechanizable, or in practice at least mechanically checkable. Every step is justified by application of some rule in an entirely syntactic process. No mathematicians have to vote on whether the resulting proof has been constructed properly.

The client of a software component will also wish to know that the two dashed arrows in Fig. 5 are "correct." Showing this is another matter entirely. The top arrow means that requirement R "is captured by" specification A. This is a somewhat vague property relating an informal behavioral description to a formal one. At best we can hope to make a believable, rigorous, but nonetheless informal argument that the relation holds. Similarly, the other dashed arrow means that program C "is expected to behave as" observation E. Again, we can hope to argue that this is so but not prove it formally. The case is based on arguments that the formal semantics defined for the implementation language match what a reader of the code intuitively expects, that the compiler-generated code also matches this expectation, and even that the hardware will not be unplugged while the client is executing the component. Some of the links in this chain are not subject to formal proof methods.

The question of the roles of verification and testing has recently been debated in the open literature with a fervor reminiscent of a religious argument; see (Fetzer, 1988) and letters to the editor in subsequent issues of *Communications of the ACM*. The spirit of the exchange is similar to one prompted a decade earlier by (DeMillo *et al.*, 1979). Several points are important here. First, verification is not a cure for bad programming. It is a formal way of capturing the informal reasoning about program behavior that a programmer must be able to undertake and to understand in any case. If one can reason informally — and correctly — about a program's behavior, then in principle that reasoning can be formalized. Equivalently, if formal verification of a program is difficult or impossible then so is rigorous informal reasoning about its behavior. The question of whether a programmer ought (in principle if not in practice) to be able to verify his/her program is therefore tantamount to the question of whether he/she ought to be able to reason about it and understand it. Few would argue that this is a superfluous or incidental activity.

Second, in the terms of Fig. 5, verification should be considered a way of certifying that C correctly implements A. It deals only with the formally-defined products of the software engineering process and formal relationships between them shown on the right side of Fig. 5. Verification cannot — and should not purport to — show that the formal specification of behavior in A captures the requirement that is intuitively stated in R, or that in actual execution E the concrete component behaves as it is coded in C.

Third, verification refers to a formal process of symbol manipulation according to well-defined proof rules associated with the formal languages used to write abstract behavioral specifications and concrete implementations. The literature contains examples of programs that have been "proved correct," only later to be shown erroneous in some respect. In the setting of Fig. 5, the reason for this discrepancy is generally that the so-called "proofs" are necessarily informal arguments about the correctness of C relative to R, not relative to A. These arguments are written in natural language and, like the informal behavioral descriptions they deal with, are ambiguous and subject to misinterpretation and misunderstanding. Such arguments by definition are *not* program verifications.

Finally, testing is usually treated as a method for checking the composite relation that E matches R. Even if a formal specification is available to test against (Gannon *et al.*, 1981; Gourlay, 1983), the best that testing can do is to check E against A. It has been noted that testing can only show the presence of errors, not their absence (Dijkstra, 1972). In fact, though, if defects are discovered by testing they may be almost anywhere: in the translation of R to A, in the implementation of A by C, in the execution of C producing E, or in the comparison of R to E. A problem detected in testing therefore does not even imply the *presence* of an error in the implementation C. On the other hand, given that C has been certified to implement A correctly (i.e., where this has been shown by verification) testing may be able to increase confidence in a conclusion that R is captured by A (Gerrard *et al.*, 1990) and/or that the actual execution E is what should be expected of C. Verification alone cannot do this.

Verification and testing therefore can be viewed as important complementary (not competing) techniques essential to the development of a mature software components industry. Advances in these areas, however, generally have not been applied to reusable software until very recently. Therefore, existing reusable components typically are accompanied by no certification of correctness of implementation. Of course certification is complicated by the lack of formal specifications, since without them it is impossible to know whether a concrete component is in fact behaving properly. Even with formal specification the problem is very difficult because the techniques necessary to verify correctness are not yet well-developed, and they generally have not been applied to programs with complex modular composition and non-trivial data structures.

5.2.1 Verification

In the same way that reusability of software justifies the cost of engineering it to be efficient and flexible, it also justifies what may turn out to be an even higher cost for verifying it. If software is to be relied upon heavily, as a reusable software component surely will be, it is vital that it be completely correct. This is a challenging issue because although there has been work on program verification for over twenty years, starting with (Floyd, 1967; Hoare, 1969), there are still few programs whose correctness has been — or could be — formally verified. The paper in which (DeMillo *et al.*, 1979) claim they can “only try to argue against verification, not blast it off the face of the earth” seems to have had a more chilling effect than the authors humbly imagined. From about the time of that paper until now there has been relatively little activity on the verification front in general.

However, the more recent work, including some related to Alphard (Shaw, 1981), Modula-2 (Ernst *et al.*, 1982), Ada/ANNA (Luckham *et al.*, 1987), and Larch/Ada with an interesting subset of Ada (Guaspari *et al.*, 1990), bears directly on the verification of reusable components. It follows the general lines of decade-earlier efforts but concentrates on modularity of proofs and programs and on examples comparable to those envisioned as implementations of non-trivial reusable components. At this point, in fact, there is every reason to be optimistic about the prospects for verification of reusable components and their clients’ uses of them, even if not arbitrary programs.

One of the main reasons for optimism is that when abstract components are specified as in Section 3, client code becomes much easier to verify because the verification factors nicely along component boundaries. If the more sophisticated components are implemented in layers on top of simpler ones, then it is relatively easy to verify each of the pieces, as contrasted with the verification of a monolithic program of similar size or complexity (Shaw, 1983; Krone, 1988). This finding is contrary to the claim of (DeMillo *et al.*, 1979) that “there is no reason to believe that a big verification can be the sum of many small verifications.” The reason verifications are comparatively simple in this setup is that the mathematical language used for specification at successive levels in the hierarchy of components changes appropriately as we move up. At the bottom the specifications may be talking about integers and cartesian products, while at higher levels they may be expressed in terms of functions, sets, strings, graphs, equivalence relations, or whatever mathematical structures are appropriate.

The fact that a concept may be described in one mathematical theory, while its content may involve lower-level concepts described in completely different theories, implies that an implementation must include a formal description of the *correspondence* relation between the abstract mathematical model that explains the concept and its concrete representation. Because they lack integrated specification languages in which to write abstract behavioral descriptions, and because they were not designed to support verification, Ada, C++, and Eiffel have no place in which to express such a correspondence. RESOLVE implementations have a syntactic slot in which the implementer must state this assertion. The proof system for RESOLVE uses the correspondence clause to elaborate the conceptual specifications of each operation into the mathematical language in which the operations in the implementation code are specified. This elaboration is done entirely at the syntactic level (something better done by machine than by a human), and it is performed only once for each concrete component (Krone, 1988).

The potential payoff of factoring in terms of simplifying presumably many proofs of client programs that instantiate a reusable concept is substantial (Musser and Stepanov, 1989). However, considerable additional work is needed before mechanical verification, or even mechanical checking of program correctness proofs, becomes practical on a large scale — even for the special case of reusable components. Some of the problem lies in the area of mathematical theorem proving, some in the area of proof rules for programming languages that support reuse. We hope that the current philosophical flap over the utility of verification does not have the same negative impact on research in this area that was observed the last time such a controversy arose.

5.2.2 Testing

Testing, like verification, has been investigated for many years, but again only recently with respect to software components like those considered here. One of the first published techniques of this kind is described by (Gannon *et al*., 1981), who introduce a method called DAISTS to test a component against its algebraic specification. The axioms are equations that define functions that are directly implemented by program operations. They include axioms for an EQUAL function. A variety of test points are chosen for each axiom and that axiom is "executed" on each one, i.e., both sides of the equation are evaluated by composing operations as required. Whether the equation holds is then checked using the Equal operation on the two results.

There are several problems with this method, some obvious and some more subtle. For example, (Gannon *et al*., 1981) note that an (erroneous) implementation of the Equal operation that does not compute the EQUAL function but always returns "true" results in tests that never find defects. More fundamental problems are noted by (Hegazy, 1989), who demonstrates how the method can be blind to implementation errors even when the Equal operation is apparently correct. Specifically, it is possible to have implementations that pass all of DAISTS tests for every axiom and for every possible test point, yet would fail if the tests included certain theorems that are derivable from the axioms using the rules of the logic of the specification language. Such implementations cannot be proved correct by verification, but no amount of testing of the axioms alone can reveal an error.

DAISTS tests compositions of operations because the effects of compositions are specified in the algebraic approach. This is at the heart of the difficulty noted above. There are also methods for testing reusable components having model-based specifications. Here the operations can be tested individually, making traditional program testing results and techniques appear more directly applicable. For example, (Liskov and Guttag, 1986) note the importance of path coverage in testing components. Other conventional test data adequacy criteria involving control flow and data flow can also be adapted for use with reusable components having model-based specifications. Surprisingly, it has been observed that there are differences between the relative theoretical power of these criteria when applied to formally specified reusable components and their relative theoretical power when applied to ordinary programs written in conventional languages (Zweben, 1989).

Several other interesting issues related to testing of reusable components are investigated by (Hegazy, 1989). For instance, there is a question of "observability": How can the value of a variable with a hidden representation be observed in evaluating the outcome of a test? In traditional testing one simply prints out the values of program variables and assumes that the language's built-in print routine works properly. For instance, a 32-bit Integer's representation is displayed in the form of a decimal number that can be interpreted as its mathematical model. Similarly, a Stack should be displayed in the form of its abstract model (a mathematical STRING). But who writes the operation that does this? How does that operation affect the Stack being displayed? Displaying the value of an exported type ordinarily is not a primary operation, but if it is coded as a secondary operation layered on top of the primary operations then errors in the primary operations or in the implementation of the display operation can manifest themselves in bizarre ways that impact the integrity of the testing procedure.

There are many other questions that need to be answered in order to make testing of reusable components possible (Hegazy, 1989). Assuming that important test cases can be identified, how can variables be driven to have the desired values? For some types that might be defined by reusable components it is not obvious how to do this. What instances of a generic component should be tested? Can the usual test data adequacy criteria be adapted to answer this question? There seem to be more questions than answers in these areas.

As noted above, testing and verification are complementary techniques for certification of correctness. They seemingly can be combined synergistically. One interesting connection is that language support for component verification and testing seem to require similar constructs. The component testing method proposed by (Hegazy, 1989) depends on two important language mechanisms of RESOLVE that permit the programmer of a concrete component to write a special *model* operation that displays the value of an exported program type in the form of its abstract mathematical model, not in terms of its internal representation. Language constraints prevent the implementation of this operation from disturbing the internal representation, which is important in testing. It turns out that the same constructs allow a programmer to mix into the code what are known as *adjunct* variables: purely mathematical variables that do not participate in program execution, but that simplify the statement of correspondences and other assertions. Adjunct variables have previously been shown to be important in verification (Krone, 1988).

6. Conclusions

Reusable software components have long appeared to be potentially attractive weapons in the war on the "software crisis." However, they have not yet proved to be as valuable as their counterparts in more mature engineering disciplines. There are a variety of non-technical and technical reasons for this phenomenon. We have reported on a number of recent inroads into answering the technical questions, which are summarized below.

In Section 2 we have explained a specific model of component-based software structure as the foundation for discussion. This model, recently dubbed the 3C reference model, emphasizes the importance of separating *concept* (what a component does) from *content* (how a component works). It further distinguishes intrinsic concept and content from *context* (environment of a component). In the 3C model there are two kinds of components: abstract and concrete, corresponding to concept and content, respectively. Each abstract component admits many different concrete components that realize its abstract behavior.

One of the reasons for postulating a specific model of software structure is the need to sketch a vision of a mature software components industry. Any long-term viable approach to software component reuse must relentlessly pursue the ultimate goal as well as intermediate milestones. A key point here is our conclusion — based on technical, economic, and legal grounds — that a software components industry eventually will be based on formal specification of abstract component behavior and the general *un*availability of source code to concrete component clients. For some programmers the absence of source code for components seems virtually impossible to comprehend. In fact, we have faced considerable resistance to the suggestion that it is as potentially beneficial as it is inevitable. There are already pockets of programmers writing commercial software for whom the absence of source code for components poses no serious problem, though.

Based on this likely long-term scenario, we have discussed in Section 3 some approaches to formal specification of functional behavior of reusable components. The two major contenders, which are usually called the algebraic and model-based approaches, are more alike than at odds. They are really differences in style. Recent developments and personal experience in teaching computer science students as well as practicing software engineers lead us to predict that the model-based style will be preferred for writing formal specifications of reusable components.

In Section 4 we have recommended both general and specific guidelines to direct designers of reusable software components toward superior abstract designs that have efficient implementations. The literature reveals a surprising variety of contradictory suggestions for all but the most general statements of what constitutes “good design.” We have compared and contrasted previous suggestions and have augmented the more useful ones with very specific corollaries. A stack module (used as an example here) is among the simplest reusable components. It is frequently disparaged as being so simple that it cannot illustrate any but the most trivial point. However, the variety of stack designs encountered in the literature — all of which are inferior to the one eventually developed in Section 4.4.3 — reveals the importance of having a “handbook of design” for reusable components. Even an apparently simple component is difficult to design for reuse. Designing a family of components that fit together comfortably is an orders-of-magnitude harder task and, we contend, cannot be done successfully without strict adherence to a set of standard design guidelines and conventions.

Finally, in Section 5 we have explored methods for factoring the context of reusable components from their abstract specifications and concrete implementations in order to support component adaptation by clients. Two powerful techniques currently are used for this purpose: genericity and inheritance. Genericity will be more useful if it is extended from its definition in, e.g., Ada and Eiffel. Inheritance, on the other hand, will be more useful if it is significantly restricted from its definition in, e.g., C++ and Eiffel. Specifically, inheritance of abstract behavior is safe and valuable under many circumstances. Inheritance of implementation code is dangerous and counterproductive with respect to modularity, information hiding, and other accepted software engineering principles. In RESOLVE we have introduced a powerful form of genericity and a method of specification inheritance (called enhancement) that together permit a vast array of reuse possibilities heretofore not feasible.

We have also briefly discussed techniques for certifying the correctness of concrete components relative to their abstract counterparts. This is one of the most important issues facing the software engineering community in general — one where there are a number of hard technical problems that remain to be solved, but also one where we envision significant progress over the next decade. Both verification and testing are crucial aspects of the certification of correctness of reusable components. We would not be surprised to see the software components industry spawn the development of a cadre of independent component certifiers whose trustworthiness and very economic survivability depend on the quality of their work.

Throughout the chapter we have also discussed the influences of programming language mechanisms on component reuse, and the influences of component reuse on programming language design. There is plenty of room for improvement in programming languages to support software component reuse. It would be a serious mistake to assume that any existing language contains “the right constructs” and that no further language work is necessary. It is tempting to denounce new languages as pipe-dreams in the face of the massive infusion of support now going into Ada, for example. However, there is already a clear recognition of Ada's weaknesses and a lively debate over what Ada-9X should look like. By the time Ada-0X is being considered a decade from now, it will probably resemble Ada only superficially and then only for reasons of upward compatibility. Some of the lessons learned from other languages in the interim surely will be applicable to the re-design of Ada.

Despite significant recent advances, then, there are still a number of important obstacles to the development a mature software components industry. We feel confident they can be adequately addressed in the next several years and that, if the non-technical impediments to reuse can be overcome in the same time frame, a rudimentary software components industry will be seen to take shape before the end of the century.

ACKNOWLEDGMENTS

We are indebted to our colleagues and our current and former students in the Reusable Software Research Group, especially Denise Byrnes, Doug Harms, Wael Hegazy, Wayne Heym, Joe Hollingsworth, Joan Krone, Allen Parrish, Tanya Pittel, Shantanu Sarkar, Murali Sitaraman (a.k.a. S. Muralidharan), Michael Stovsky, and Lonnie Welch. The efforts and suggestions of the many students who have served as "guinea pigs" for some of our ideas in CIS 680 at Ohio State and in CS 4030 at AT&T Bell Laboratories are appreciated, too. We also acknowledge the assistance of many other people, including but not limited to Sam DeFazio, Steve Edwards, George Ernst, Stanley Habib, Chua-Huang Huang, D. N. Jayasimha, Doug Kerr, Wolfgang Kuechlin, Tim Long, Jorge Ochoa-Lions, P. Sadayappan, Mark Simos, Neelam Soundararajan, and Will Tracz. They have certainly helped us to better understand reusable software components and how to explain them. Finally, we thank the National Science Foundation and the Applied Information Technologies Research Center for financial support.

REFERENCES

Ada Joint Program Office (1983). "Reference Manual for the Ada Programming Language." ANSI/MIL-STD-1815A-1983, U.S. Government Printing Office, Washington, DC.

Apple Computer (1985). "Inside Macintosh — Volumes I-V." Addison-Wesley, Reading, MA.

Bentley, J.L. (1982). "Writing Efficient Programs." Prentice Hall, Englewood Cliffs, NJ.

Berard, E.V. (1987). "Creating Reusable Ada Software." EVB Software Engineering, Inc., Frederick, MD.

Biggerstaff, T.J., and Perlis, A.J., eds. (1989). "Software Reusability — Volume I: Concepts and Models, Volume II: Applications and Experience." Addison-Wesley, New York.

Birtwistle, G., Dahl, O-J., Myrhaug, B., and Nygaard, K (1973). "Simula Begin." Studentliteratur, Lund, and Auerbach, New York.

Bjørner, D., Jones, C.B., Mac an Airchinnigh, M., and Neuhold, E.J., eds. (1987). "VDM '87: VDM — A Formal Method at Work." Springer-Verlag, Berlin.

Boehm, B.W. (1987). Improving software productivity. *Computer* **20** (9), 43-57.

Booch, G. (1987). "Software Components with Ada." Benjamin/Cummings, Menlo Park, CA.

Computer Science and Technology Board (1990). Scaling up: a research agenda for software engineering. *Comm. ACM* **33** (4), 281-293.

Cox, B.J. (1986). "Object Oriented Programming: An Evolutionary Approach." Addison-Wesley, Reading, MA.

DeMillo, R.A., Lipton, R.J., and Perlis, A.J. (1979). Social processes and proofs of theorems and programs. *Comm. ACM* **22** (5), 271-280.

Dijkstra, E.W. (1972). The humble programmer. *Comm. ACM* **15** (10), 859-866.

Edwards, S. (1990). The 3C model of reusable software components. In *Third Ann. Workshop: Methods and Tools for Reuse*, Syracuse Univ. CASE Center.

Embley, D., and Woodfield, S. (1988). Assessing the quality of abstract data types written in Ada. In *Proc. Intl. Conf. on Softw. Eng.*, IEEE, 144-153.

Ernst, G.W., Navlakha, J.K., and Ogden, W.F. (1982). Verification of programs with procedure-type parameters. *Acta Inf.* **2** (4), 522-543.

Fairley, R. (1985). "Software Engineering Concepts." McGraw-Hill, New York.

Feldman, M.B. (1988). "Data Structures with Modula-2." Prentice Hall, Englewood Cliffs, NJ.

Fetzer, J.H. (1988). Program verification: the very idea. *Comm. ACM* **31** (9), 1048-1063

Floyd, R. (1967). Assigning meanings to programs. In *Proc. AMS Symp. on Appl. Math.,* AMS, 119-131.

Gannon, J.D., McMullin, P., and Hamlet, R. (1981). Data-abstraction implementation, specification, and testing. *TOPLAS* **3** (3), 211-223.

Gannon, J.D., and Zelkowitz, M.V. (1987). Two implementation models of abstract data types. *Comp. Lang.* **12** (1), 21-25.

Gautier, R.J., and Wallis, P.J.L., eds. (1990). "Software Reuse with Ada." Peter Peregrinus Ltd., London.

Gerrard, C.P., Coleman, D., and Gallimore, R.M. (1990). Formal specification and design time testing. *IEEE Trans. on Softw. Eng.* **16** (1), 1-12.

Goguen, J. (1984). Parameterized programming. *IEEE Trans. on Softw. Eng.* **SE-10** (5), 528-543.

Goldberg, A., and Robson, D. (1983). "Smalltalk-80: The Language and its Implementation." Addison-Wesley, Reading, MA.

Gourlay, J.S. (1983). A mathematical framework for the investigation of testing. *IEEE Trans. on Softw. Eng.* **SE-9** (6), 686-709.

Gu, J., and Smith, K.F. (1989). A structured approach for VLSI circuit design. *Computer* **22** (11), 9-22.

Guaspari, D., Marceau, C., and Polak, W. (1990). Formal verification of Ada programs. *IEEE Trans. on Softw. Eng.* **16** (9), 1058-1075.

Guttag, J.V., Horning, J.J., and Wing, J.M. (1985). The Larch family of specification languages. *IEEE Software* **2** (5), 24-36.

Guttag, J.V., and Horning, J.J. (1986a). Report on the Larch Shared Language. *Sci. of Comp. Prog.* **6**, 103-134.

Guttag, J.V., and Horning, J.J. (1986b). A Larch Shared Language handbook. *Sci. of Comp. Prog.* **6**, 135-157.

Harms, D.E. (1990). "The Influence of Software Reuse on Programming Language Design." Ph.D. diss., Dept. of Comp. and Inf. Sci., Ohio State Univ., Columbus, OH.

Hegazy, W.A. (1989). "The Requirements of Testing a Class of Reusable Software Modules." Ph.D. diss., Dept. of Comp. and Inf. Sci., Ohio State Univ., Columbus, OH.

Hibbard, P., Hisgen, A., Rosenberg, J., and Sherman, M. (1983). Programming in Ada: examples. In "Studies in Ada Style." P. Hibbard, A. Hisgen, J. Rosenberg, M. Shaw, and M. Sherman, eds. Springer-Verlag, New York, 35-101.

Hoare, C.A.R. (1969). An axiomatic basis for computer programming. *Comm. ACM* **12** (10), 576-581.

Hoare, C.A.R. (1983). Hints on programming language design. In "Programming Languages: A Grand Tour." E. Horowitz, ed., Computer Science Press, Rockville, MD.

IEEE (1984). Special issue on software reusability, *IEEE Trans. on Softw. Eng.* **SE-10** (5).

Jones, W.C. (1988). "Data Structures Using Modula-2." John Wiley & Sons, New York.

Krone, J. (1988). "The Role of Verification in Software Reusability." Ph.D. diss., Dept. of Comp. and Inf. Sci., Ohio State Univ., Columbus, OH.

LaLonde, W. R. (1989). Designing families of data types using exemplars. *TOPLAS* **11** (2), 212-248.

Latour, L., Wheeler, T., and Frakes, W. (1990). Descriptive and predictive aspects of the 3Cs model: SETA1 working group summary. In *Third Ann. Workshop: Methods and Tools for Reuse*, Syracuse Univ. CASE Center.

Liskov, B., and Guttag, J. (1986). "Abstraction and Specification in Program Development." McGraw-Hill, New York.

Liskov, B.H., and Zilles, S.N. (1975). Specification techniques for data abstractions. *IEEE Trans. on Softw. Eng.* **SE-1** (1), 7-19.

Liskov, B.H., Atkinson, R., Bloom, T., Moss, E., Schaffert, J.C., Scheifler, R., and Snyder, A. (1981). "CLU Reference Manual." Springer-Verlag, New York.

Livintchouk, S.D., and Matsumoto, A.S. (1984). Design of Ada systems yielding reusable components: an approach using structured algebraic specification. *IEEE Trans. on Softw. Eng.* **SE-10** (5), 544-551.

London, R.L., and Milsted, K.R. (1989). Specifying reusable components using Z: realistic sets and dictionaries. *Softw. Eng. Notes* **14** (3), 120-127.

Luckham, D., von Henke, F.W., Krieg-Brückner, B., and Owe, O. (1987). "ANNA: A Language for Annotating Ada Programs." Springer-Verlag, Berlin.

Martin, J.J. (1986). "Data Types and Data Structures." Prentice Hall, Englewood Cliffs, NJ.

McIlroy, M.D. (1976). Mass-produced software components. In "Software Engineering Concepts and Techniques." J.M. Buxton, P. Naur, and B. Randell, eds. Petrocelli/Charter, Brussels, 88-98.

Meyer, B. (1985). On formalism in specification. *IEEE Software* **2** (1), 6-26.

Meyer, B. (1986). Genericity versus inheritance. In *OOPSLA '86 Proc.*, ACM, 391-405.

Meyer, B. (1988). "Object-oriented Software Construction." Prentice Hall, Englewood Cliffs, NJ.

Muralidharan, S. (1989). On inclusion of the private part in Ada package specifications. In *Proc. 7th Ann. Natl. Conf. on Ada Tech.*, ANCOST, Inc., 188-192.

Muralidharan, S., and Weide, B.W. (1990). Should data abstraction be violated to enhance software reuse? In *Proc. 8th Ann. Natl. Conf. on Ada Tech.*, ANCOST, Inc., 515-524.

Musser, D.R., and Stepanov, A.A. (1989). "The Ada Generic Library: Linear List Processing Packages." Springer-Verlag, New York.

Parnas, D.L. (1972). On the criteria to be used in decomposing systems into modules. *Comm. ACM* **15** (12), 1053-1058.

Pittel, T.S. (1990). “Pointers in RESOLVE: Specification and Implementation.” M.S. thesis, Dept. of Comp. and Inf. Sci., Ohio State Univ., Columbus, OH.

Prieto-Diaz, R. (1987). Domain analysis for reusability. In *Proc. COMPSAC ’87*, IEEE, 23-29.

Prieto-Diaz, R. (1990). Domain analysis: an introduction. *Softw. Eng. Notes* **15** (2), 47-54.

Raj, R.K. (1990). Code inheritance considered harmful. In *Third Ann. Workshop: Methods and Tools for Reuse*, Syracuse Univ. CASE Center.

Raj, R.K., and Levy, H.M. (1989). A compositional model for software reuse. *Comp. J.* **32** (4), 312-322.

Samuelson, P. (1988). Is copyright law steering the right course? *IEEE Software* **5** (5), 78-86.

Shaw, M., ed. (1981). “ALPHARD: Form and Content.” Springer-Verlag, New York.

Shaw, M. (1983). The impact of abstraction concerns on modern programming languages. In “Studies in Ada Style.” P. Hibbard, A. Hisgen, J. Rosenberg, M. Shaw, and M. Sherman, eds. Springer-Verlag, New York, 7-32.

Sitaraman, M. (1990). “Mechanisms and Methods for Performance Tuning of Reusable Software Components.” Ph.D. diss., Dept. of Comp. and Inf. Sci., Ohio State Univ., Columbus, OH.

Snyder, A. (1986). Encapsulation and inheritance in object-oriented systems. In *OOPSLA ’86 Proc.*, ACM, 38-45.

SofTech, Inc. (1985). “ISEC (U.S. Army Information Systems Engineering Command) Reusability Guidelines.” 3285-4-247/2, Waltham, MA.

Spivey, J.M. (1989). “The Z Notation: A Reference Manual.” Prentice-Hall, Englewood Cliffs, NJ.

Stroustrup, B. (1986). “The C++ Programming Language.” Addison-Wesley, Reading, MA.

Stubbs, D.F., and Webre, N.W. (1987). “Data Structures with Abstract Data Types and Modula-2.” Brooks/Cole, Monterey, CA.

Tomijima, A.U. (1987). How Japan’s recently amended copyright law affects software. *IEEE Software* **4** (1), 17-21.

Tracz, W. (1987). Reusability comes of age. *IEEE Software* **4** (4), 6-8.

Tracz, W. (1990a). Where does reuse start? *Softw. Eng. Notes* **15** (2), 42-46.

Tracz, W. (1990b). The three cons of software reuse. In *Third Ann. Workshop: Methods and Tools for Reuse*, Syracuse Univ. CASE Center.

Tracz, W., and Edwards, S. (1989). Implementation issues working group report. In *Reuse in Practice Workshop*, SEI, Pittsburgh, PA.

Wallace, D.R., and Fujii, R.U. (1989). Software verification and validation: an overview. *IEEE Software* **6** (3), 10-17.

Weiser, M. (1987). Source code. *Computer* **20** (11), 66-73.

Wing, J.M. (1987). Writing Larch interface language specifications. *TOPLAS* **9** (1), 1-24.

Wing, J.M. (1990). A specifier's introduction to formal methods. *Computer* **23** (9), 8-24.

Zweben, S.H. (1989). "Testing Formally Specified Data-Oriented Modules Using Program-Based Test Data Adequacy Criteria." OSU-CISRC-8/89-TR39, Dept. of Comp. and Inf. Sci., Ohio State Univ., Columbus, OH.