SOFTWARE REUSABILITY

The software industry is poised to become the biggest industry in the world. Demands for timely production of high-quality complex software systems have become more acute than ever. In discussing the heightened productivity and quality crisis, Gibbs notes, "Studies have shown . . . the average software development project overshoots its schedule by half; large projects generally do worse. And some three-quarters of all large systems are 'operating failures' that either do not function as intended or are not used at all" (1). Explaining the economic and possibly life-threatening consequences of this crisis, Gibbs concludes, "If we are ever going to lick this software crisis, we're going to have to stop this hand-to-mouth, every-program-from-the-ground-up preindustrial approach."

Software reusability is an essential mechanism for improving both software productivity and quality and must be an inherent part of any successful software engineering process (2–7). The objective of research in software reusability is to build "high-quality" software-related artifacts that are more widely applicable than they otherwise would be and can be reused with less effort than otherwise needed. Examples of reusable software artifacts include descriptions of communication protocols; operating system templates that are instantiated differently using different parameters for different installations; descriptions of patterns and styles used in explaining a class of software artifacts; concept definitions that are generally applicable in solving a class of problems; descriptions of algorithms; general-purpose software modules and packages; and metalevel descriptions.

PRODUCTS AND PROCESSES

Although the nature and potential of reusable software artifacts vary significantly, some governing principles apply. The first of these is the observation that reusing a software artifact has potential for significant gains only if the effort invested in conceptualizing and developing a high quality artifact is leveraged. For example, a "cut and paste" reuse process that merely saves keystrokes has little value. A corollary to this observation is that reusing poorly designed artifacts (products) has a negative impact on software productivity and quality, regardless of the process. In other words, the central problems in software reusability are in designing "high-quality" software *products* in the first place and then establishing *processes* that maximize reuse of effort invested in designing the products.

In general, reusable product developers and clients are different people or organizations. Therefore, in a reusability process, it is useful to classify the client effort to reuse an artifact into three core parts:

- *Understanding.* The effort involved in understanding the existing artifact in sufficient detail to reuse;
- *Adaptation.* The effort involved in modifying the artifact so it is suitable for the new situation; and
- *Reasoning.* The effort involved in ensuring that the adapted artifact captures the intent or behaves as expected, and continues to be of high quality.

The illustration in Fig. 1 contains a "part-based" description of some artifacts, and a cut-and-paste adaptation of one of its parts. In the example in Fig. 1, the benefits of reuse are minimal, though most of the original artifact is reused. Because of the assumed coupling among the descriptions of the parts of this artifact (which is inevitable in the absence of abstraction, as explained later), any change requires *global understanding* of the existing artifacts and *global reasoning* of the modified artifacts to ensure that the change produces exactly the desired impact. In this scenario, whereas the potential cost of understanding dominates the effort in developing the artifact in the first place, the need for new global reasoning makes it

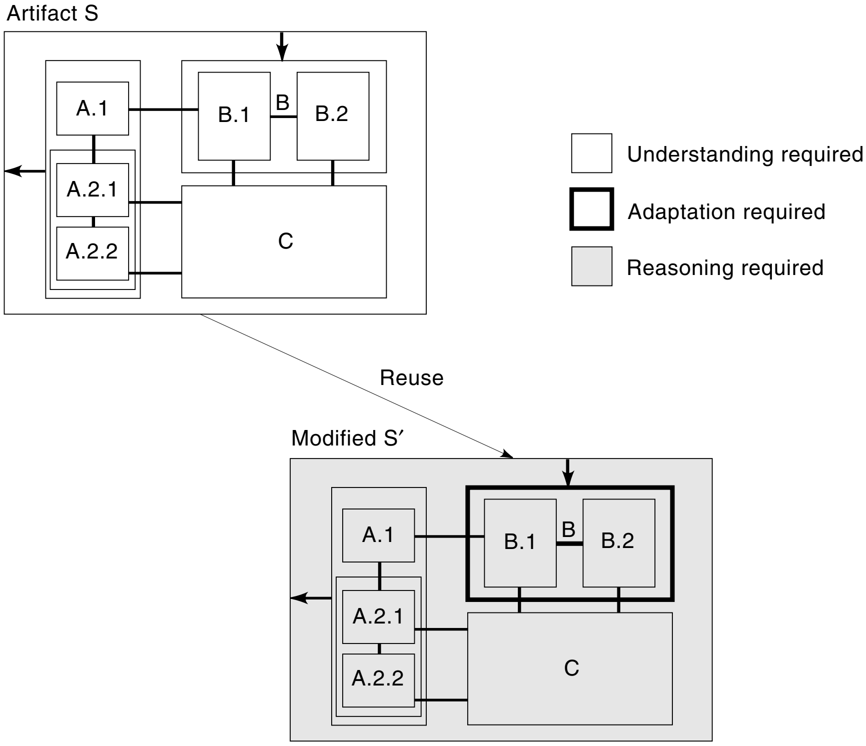

Figure 1. The need for global understanding and reasoning in reuse without abstraction.

impossible to reuse the significant effort already invested in verifying (or validating) the existing effort.

Although the example highlights the drawbacks of a cut-and-paste reuse process, the process itself is only as good as the design and description(s) of the product allow it to be, as explained in the next subsection. The inherent nature of some artifacts forces their only descriptions to be detailed in terms of these parts and thus makes it impossible to avoid global understanding and reasoning. However, for others, *abstract* descriptions that are independent of their parts can be developed to enable *localized understanding and reasoning* and, hence, ease of reuse. Reuse of the latter kinds of artifacts is the focus of this article.

DESIGN FOR REUSE

An artifact is designed for reuse along a given *dimension* if the cumulative effort for understanding, adaptation, and reasoning needed for reuse is kept minimal and proportional to the required adaptation. Dimension here includes functional and nonfunctional attributes and software, hardware, system, and other environment-related aspects. In designing software or any other engineering artifact for ease of use and adaptation, four basic ideas come into focus: compartmentalization, hierarchical composition, generalization, and abstraction. Of these, the first two are most widely applicable, but the last two have the most impact on software reuse.

Compartmentalization and Hierarchical Composition

The principles of compartmentalization and hierarchical composition (together termed modularization) are and should be applied in describing all nontrivial artifacts. Figure 1 shows a part-based description of some artifact. It is the *sum* of the descriptions of its 3 parts, namely, A, B and C, combined in a particular way. Similarly, descriptions A and B have been further decomposed into parts. In a complete description of an artifact, each part is decomposed hierarchically until the descriptions are atomic, that is, descriptions that are universally understood by the intended audience.

The artifact described in Fig. 1 might be a general document such as this article on software reusability which is divided into sections, subsections, paragraphs, sentences, and words. To understand this article, ultimately every word must be understood. Alternatively, the description might be a requirements definition for a software system or a programming language description (or “code”) for a software system, divided and subdivided into procedures or objects. In programming language descriptions, the primitives are built-in objects such as Integers, Records, or Arrays and permissible operations on these objects.

Decomposition into parts is essential for cognitive simplification. It also provides an organization that is important in understanding the artifact described. Different techniques for decomposition lead to different parts and organizations. A code artifact, for example, might be decomposed by structured analysis and design techniques or object-oriented techniques. Through such modularization, sometimes, changes needed for reuse are localized within a few parts. However, modularization of a description into parts has only a limited impact on reuse cost, because the effort in understanding the artifact to adapt it and the effort required in reasoning about the adaptation remain global efforts. Changing a sentence in this subsection of this article for its content, for example, can have global ramifications as it might contradict statements in other parts of this article. Similarly, with modularization alone, understanding and reasoning of a program requires breaking it down to primitive objects and operations such as those on Integers (8,9). In more general terms, to understand and adapt a part-based description such as in Fig. 1, the part to be adapted and the part-based descriptions of every artifact with which it is coupled must be understood, that is, every atomic part of the artifact. In summary, compartmentalization and hierarchical decomposition by themselves do not reduce the complexity of understanding or reasoning.

Generalization

The objective of generalization is to anticipate the circumstances in which an artifact might be reused and then to design the artifact so that reuse costs are minimal in those circumstances. Suitable generalization is a challenging design problem. *Parameterization* is one fundamental mechanism for generalization (10). Parameters allow tuning an artifact for the needs of a specific application without otherwise modifying the artifact. Classical examples of parametric adaptation include software system installations tailored for local environments and generic software modules. Using the same approach, it is possible to parameterize requirements definition documents or other artifacts for easy adaptation. Parameterization, however, is only one factor in generalization. Design techniques, such as those based on objects for code artifacts (11,12), significantly influence the generality of an artifact.

To limit the complexity and cost of generalization in current practice, some artifacts are designed to be reused only within a vertical domain. For others, the scope is narrowed even further by considering only product lines within specific organizations. Reuse of an artifact during relatively routine maintenance of the same product in the updated versions of the product is an example of the case of reuse where the scope is further narrowed.

Generalization reduces the cost of adaptation to new environments significantly and simplifies understanding along the dimension of parameters. Generalization does not however eliminate the need for global understanding or reasoning, in the absence of abstraction.

Abstraction

Abstraction is an essential cognitive tool in human understanding and reasoning. Good abstraction allows us to establish suitable and simple mental models for software artifacts (13). Abstraction is what makes it easier to understand and use physical devices such as batteries, mowing machines, and televisions, though their internal part-based descriptions might be extremely complex. Similarly, for some software-related artifacts, it is possible to have two (or more) *independent* descriptions, one of which is more abstract and is not based on the parts of the artifact. Whereas the lower level description might require an understanding of its parts, the high level description would not. Free of details and much simpler, the higher level description is a suitable "cover story" that reveals exactly and only what a user of the artifact needs to know. Abstract descriptions are typically what make physical devices, such as batteries and televisions, easy to use. To understand and use them, it is not essential to understand their internal part-based description.

If each part in the description in Fig. 1 had a separate and independent abstract explanation, only local understanding and reasoning would be required as illustrated in Fig. 2. The figure shows a window into a much larger structure. In Fig. 2, presented in a variant of the 3C model that is widely used in the reuse community (14), ovals denote abstract descriptions (concepts), and rectangles denote more concrete, part-based descriptions (contents). The contexts, within which the concepts and contents are described, are not explicitly shown. Thin arrows from a part-based description to abstract descriptions denote an "is composed of" relationship and thick arrows from abstract to concrete descriptions denote an "is a cover story of" relationship. There may be multiple cover stories for the same concrete description and there may be multiple concrete descriptions for the same cover story. If there are multiple interchangeable or plug-compatible concrete descriptions for abstraction B, for example, they can be switched without affecting, and without a need to reverify artifacts that are based on B.

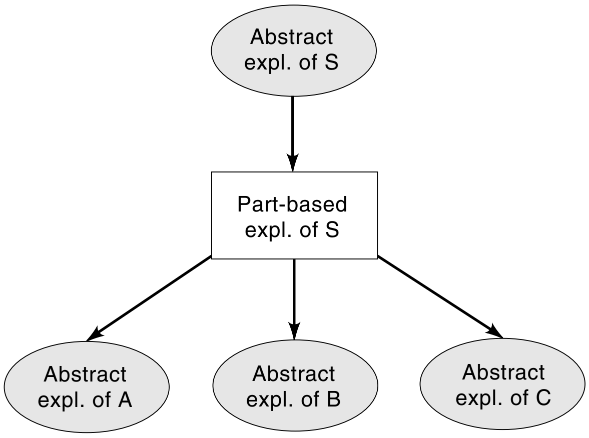

Figure 2. The role of abstraction in localizing understanding and reasoning.

The ovals in Fig. 2 are usually called specifications, and the rectangles are called implementations when code artifacts are described. Introduction of two (or more) descriptions of an artifact, essential for abstraction, also makes it essential to *verify* that the different descriptions are consistent. It must be shown that the abstract description is indeed a suitable cover story for the lower level description. When abstract descriptions are available, cost-effective local verification is possible: To verify that the part-based description in Fig. 2 satisfies its abstract description, only abstract descriptions of A, B, and C need to be understood and used (8,15–17). There is no need to understand any other descriptions or to break the reasoning down to the level of primitive descriptions or objects. In other words, abstract descriptions and techniques for localized understanding and reasoning based on those descriptions offer the most significant reuse benefits as summarized in Table 1.

In a *black box* reuse process, only the (possibility, parameterized) abstract description of an artifact needs to be understood and adapted. Because no changes are made to the descriptions, significant effort invested in verification activity is reused. Even when a *white box* reuse process is employed to alter the part-based description of the artifact in Fig. 2 directly, during evolution, maintenance, or reuse, only the abstract descriptions of its parts A, B, and C need to be under-

Table 1. Role of Abstraction in Localizing Understanding and Reasoning

	Adaptation	Understanding and Reasoning
Compartmentalization and hierarchical composition	Possibly localized	Global understanding and reasoning required. Provide cognitive simplification.
+Generalization	Significant savings for a class of adaptations	Global understanding and reasoning required. Provides cognitive simplification.
+Abstraction	Significant savings	Only local understanding and reasoning required. Also, essential to create suitable mental models.

stood if those descriptions are not altered. However, white box reuse, in general, is significantly more expensive than black box reuse and should be used only where that is inevitable (18).

Artifacts, such as algorithms, network communication protocols, and software components and systems, are typically described in detail in multiple layers in formal (programming) languages to enable effective communication with the computer. For such artifacts, multiple levels of abstract descriptions are essential. For some artifacts, however, such as for a "well-written and minimal" document defining functional requirements, there are no two, separate, possible descriptions. If every part in the document is important and is a requirement, then there is no possible simplification. The document, of course, should be compartmentalized and hierarchically composed to provide a suitable organization, but an understanding of the artifact ultimately requires understanding each part.

Artifacts, such as requirements definition documents, that allow only a single part-based description typically arise in the earlier stages of software life cycle and provide substantial reuse gains, but within a very limited scope. Code artifacts, for example, permit independent abstract descriptions that are more suitable for widespread reuse. If their different descriptions are suitably designed, they lead to significant reuse of efforts invested in their entire life cycle. When an operating system (which has abstract and part-based internal descriptions) is installed in different environments or application systems are ported, reuse benefits include the efforts invested in its design, implementation, verification, and validation.

ELEMENTS OF REUSABLE SOFTWARE DESIGN

To illustrate the variety of technical issues that must be addressed in designing a software artifact for reuse, this section considers a detailed example: abstract description of a reusable software component suitable for "prioritizing" a collection of items based on some priority ordering. The two fundamental reuse issues in designing a description, such as this, are generalization and abstraction:

The component must be properly generalized so that it can be used in a variety of applications without modifications, thus permitting reuse of the effort invested in designing, developing, and verifying the component and all its associated artifacts.

The interface must have a proper conceptualization that provides a suitable abstraction to understand and reuse the component easily, without understanding details of how it might be implemented.

One possible interface and implementation for the prioritization problem is a typical procedure that captures "batch sorting." Here, the entries to be sorted are assumed to be in a collection, and the interface provides a single sort operation that reorders the collection so that it is sorted. Although a procedural interface for batch sorting is useful in a number of applications, it can be generalized and its reusability is enhanced along at least two dimensions as explained in the following subsection.

Generalization

Interface Generalization through Object-Based Design. To enhance reusability, related classes of applications, where "prioritization" is useful, must be identified. Such anticipation is critical in designing software for reuse. Even when unanticipated uses of a concept arise, well-designed interfaces permit handling of such situations with maximal reuse of effort. For prioritization, the following alternative classes of application are useful to consider (19):

1. All entries that must be "ordered" are known a priori and all of them must be ranked.
2. All entries to be ordered are known a priori, but only an arbitrary subset of the prioritized items is needed.
   (a) some best k of inserted entries are needed; and
   (b) some best $k1$ and worst $k2$ of inserted entries are needed; and
3. Entries to be ordered are not all known in advance. After some have been obtained from a collection in prioritized order, additional entries may be added to the collection for subsequent removal.

Although it is indeed possible to use a batch sorting procedure in all of the previous applications, it is efficient only for applications in class I. Performance penalties make a typical procedural interface (and its implementations) unacceptable for solving problems in classes II and III. The need to overcome the performance barrier and thus enhance reusability is a primary motivation for "recasting" large-effect operations, such as batch sorting, and developing an object-based concept for prioritization (12).

An object-based component for the present problem provides a Prioritizer object and operations to manipulate these objects. Operations include Insert, that permits new entries to be added one at a time, and Remove_Next, that permits extraction of items in "order," one at a time. With such a component, a user of the object can remove some or all items in order, by making a suitable number of calls to Remove_Next. The user can also interleave calls to Insert and Remove_Next. More importantly, using different plug-compatible implementations of the same interface, a client can reuse Prioritizers to solve problems in any of the previous three classes of applications with optimal performance.

Generalization through Parameterization. Clearly, the idea of prioritization is independent of the entries that need to be prioritized. Therefore, another dimension of generalization is possible by *parameterizing* the type of entries to be prioritized. One design of containers, such as the prioritizers, includes a suitable upper bound on the number of items that are prioritized. These two parameters must be considered in developing general concepts of any container.

When generalizing a container, such as a prioritizer, to contain arbitrary types of entries, a fundamental design question arises: Should the Insert operation make a copy of the entry inserted into the prioritizer or should it not? If a copy is inserted, then the caller retains the inserted entry. If no copy is made, then the entry is consumed by the prioritizer, and the entry is no longer available to the caller. Discussing this design issue (i.e., whether inserted and removed items should be copied), that arises in the design of every container object, Harms and Weide (20) conclude that "copying" objects of arbitrarily complex types is expensive and leads to significantly inferior performance. Hence, Insert operations should "consume" and not copy the inserted entries. In most common uses of containers, clients do not need to retain copies of entries. In the few cases where they need to retain copies, they can make an explicit copy of the entry before inserting it in the container.

Instead of assignment and copying, Harms and Weide suggest a Swap operation (20) as the basic data movement operation on objects. By representing complex objects indirectly through references (pointers), the swap procedure is implemented so that it exchanges these references in a constant time regardless of the sizes of the referenced objects. Though references are used in implementation, swapping is "abstractly" understood as exchanging values of two objects, and thus the users are freed from complex reference-based understanding that would be required if pointers are specified to be copied explicitly.

Unlike the issues discussed previously which are general to the design of all reusable containers, aspects of parametric generalization specific to the concept under consideration are usually much harder to identify and incorporate. In the case of Prioritizer objects, it is possible to generalize the "order" in which the entries are prioritized. This ordering clearly depends on what types of items are prioritized. Even for a given type Entry, there might be multiple ways to prioritize. For example, in prioritizing planes waiting to take off from an airport, the prioritization parameter may be the scheduled take-off times for the planes, the destinations of the planes, the capacities of the planes, or a combination that depends on the actual circumstances. It is, therefore, necessary to design the object interface to be flexible with regard to the ordering.

Parameterized Prioritizer Objects in C++

Figure 3 contains Prioritizer_Template, a C++ class template for Prioritizer objects. It has been designed following a variant of the RESOLVE/C++ discipline for reusable software design (19,21). Following the principles of this discipline, a swap operator ("&=") is included and "built-in" assignment operator is precluded. The client-supplied Entry class must also have a swap operator. The class Entry_Compare_Capability must provide an operation to compare two entries. Though we have used C++ to illustrate implementation details in this article, other object-based languages are also appropriate. RESOLVE/Ada discipline and the more general principles for following the RESOLVE discipline for popular programming languages are detailed in (21). Adherence to the discipline results in components that are adaptable, and hence easy to reuse and evolve.

Clearly, each operation in Fig. 3 needs a behavioral explanation. But how should the explanations be phrased to a

```
template <class Entry, class Entry_Compare_Capability, int Max_Size>
// Entry_Compare_Capability should provide Entry Compare operation
class Prioritizer_Template {
public:

  Prioritizer_Template () {};
  virtual ~Prioritizer_Template () {};

  /* swap operator &= to be added in each implementation */
  /* virtual void operator &= (Prioritizer_Template& rhs) = 0; */
  virtual void Insert (Entry& x) = 0;
  virtual void Change_Phase () = 0;
  virtual void Remove_Next (Entry& x) = 0;
  virtual void Remove_Any (Entry& x) = 0;
  virtual Integer Total_Entry_Count_Of () = 0;
  virtual Boolean Is_In_Insertion_Phase () = 0;
  virtual void Clear () = 0;

private:
  /* Implicit assignment and copy constructor are prohibited */

  Prioritizer_Template& operator = (const Prioritizer_Template& rhs);
  Prioritizer_Template (const Prioritizer_Template& m);
};
```

Figure 3. A parameterized prioritizer object interface in C++.

```
template <class Entry, class Entry_Compare_Capability, int Max_Size>
class Prioritizer_Template_1: public
Prioritizer_Template <Entry, Entry_Compare_Capability, Max_Size>
{
public:

/* A prioritizer object has three parts: an array ‘contents’, an Integer
‘size’, and a Boolen flag ‘filling’. The elements of the object are
stored in the contents from locations 0 to size - 1.

Between 0 and size - 1, the array always remains sorted based on the
ordering in class Entry_Compare_Capability. The next smallest Entry is
in location size - 1. */

  Prioritizer_Template_1 ();
/* Initializes size to 0 and filling to true. */

  virtual ~Prioritizer_Template_1 ();
  virtual void operator &= (Prioritizer_Template_1& rhs);

  virtual void Insert (Entry& x);
/* Inserts the new Entry in the right place into the array that is kept
sorted */

  virtual void Change_Phase ();
/* Toggles filling. */

  virtual void Remove_Next (Entry& x);
/* Returns the smallest element of array from location size - 1 */

  virtual void Remove_Any (Entry& x);
/* Returns the (smallest) element of array from location size - 1 */

  virtual Boolean Is_In_Insertion_Phase ();
/* Returns true iff the filling flag is true. */

  virtual Integer Total_Entry_Count_Of ();
/* Returns size - the number of elements in the prioritizer */

  virtual void Clear ();
/* Initializes size to 0 and filling to true. */

private:
  /* Implicit assignment and copy constructor are prohibited */
  Prioritizer_Template_1& operator = (const Prioritizer_Template_1&
rhs);
  Prioritizer_Template_1 (const Prioritizer_Template_1& m);

  /* Internal representation */
  Boolean filling = false;
  Integer size = 0;
  Entry contents[Max_Size];
};
```

Figure 4. Implementation-based explanation of Prioritizer_Template.

user? This subsection contains explanations based on the objects (or parts) that are “internal” to the implementations and ramifications of using such explanations.

Explanation of Operations. Figure 4 shows Prioritizer_Template_1, a part-based interface explanation for one (but not necessarily the most desirable) implementation that inherits and provides the same interface as the Prioritizer_Template. The code for the methods (operations) are not shown, but it can be deduced from the explanations.

The understanding, reasoning, and usage of the object certainly is based on the explanations provided about the operations. For example, there is no need for a client of this object to call Change_Phase or Is_In_Insertion_Phase operations. In addition, the client may use operations Remove_Next and Remove_Any interchangeably because they have identical explanations. For example, to sort a queue q containing planes waiting to land based on some priority ordering, the client might create an instance of a Prioritizer object p with that ordering and use it as illustrated in Fig. 5. In the figure, an

```
   /* V.contents holds the items to be sorted */
   Clear (p);
   while (q.Length_Of() > 0) {
      q.Dequeue(x);
      p.Insert(x);
   };

   /* p.contents is a sorted array, based on the ordering used in
instantiating p. It contains exactly the items of the
initial queue array q.contents. q.length = 0 */

   while (p.Total_Entry_Count_Of() > 0) {
      p.Remove_Any(x);
      q.Enqueue(x);
   };

   /* q.contents is sorted.
p.size = 0 */
```

Figure 5. An example code that uses Prioritizer_Template_1.

object of type Queue is assumed to have at least an array of Entries named 'contents' and an Integer named 'length' as its parts. Dequeue, Enqueue, and Length_Of are assumed to be basic Queue object operations.

Multiple Interchangeable Implementations and Reuse. A basic form of adaptation is replacing a component with another that is plug-compatible. Figure 6 shows (explanation of) some operations of Prioritizer_Template_2, an alternative implementation of Prioritizer_Template. The C++ parts of the objects in Figs. 4 and 6 are identical. Only the explanations of operations that are different are shown. These are but two implementations of Prioritizer_Templates. Several others are discussed in (19). In general, for a given problem, such multiple implementations exist and have interesting performance tradeoffs (22).

Now suppose that we want to reuse Queue_Sort_Capability and the same objects in Fig. 1 for a different application, except that we want to replace Prioritizer_Template_1 with Prioritizer_Template_2. Given below are the steps in this reuse process:

- Understand the explanations of objects that are used, i.e., Plane_Info, Queue_Template_1, and Prioritizer_Template_1 (from Fig. 4).
- Understand the code for Queue_Sort_Capability.
- Adapt the code for Queue_Sort_Capability by changing the declaration of Prioritizer object from Prioritizer_Template_1 to Prioritizer_Template_2.
- Reason that the modified code for Queue_Sort_Capability works correctly, based on the explanation of Prioritizer object from Prioritizer_Template_2.

The last step shows that the code in Fig. 5 with the previous adaptation, though compiles without any errors, is *not correct,* that is, will not sort the queue. Two changes are needed to make the code work correctly in terms of the explanation of Prioritizer_Template_2:

- Call Change_Phase operation in between the loops to create a heap.
- Call Remove_Next instead of Remove_Any in the second loop so that the smallest element from the heap is returned.

These modifications, in turn, require reverification of the code for Queue_Sort_Capability. In other words, to reuse the code for Queue_Sort_Capability, significant effort is involved in understanding the existing artifact, reunderstanding and reasoning on the basis of the new explanation, though actual adaptation itself is minimal. This problem does not disappear even if an implementation of Prioritizer_Template is made a module-level parameter to the capability (23), thus requiring no changes to the code for adaptation because the explanations of different Prioritizer implementations differ, and parameterization cannot solve that problem.

This example confirms that object-based design, even with the best use of mechanisms such as templates and inheritance, only minimizes the cost for adaptation, because objects without abstraction must be ultimately understood only in terms of primitive programming objects as illustrated in Fig. 7. The only reason why reasoning stops with built-in objects, such as Integers and Records, is because they have well-understood mathematical integers and Cartesian products as their models.

Conceptualization

To minimize the cost to reuse objects, such as prioritizers, an abstract explanation or conceptualization (well-designed interface specification) is needed. In developing an abstract description, it is essential to use a mathematical model of the collection that differs from the way the collection is actually represented in a programming language. Arrays and lists are examples of representations for the way entries in a collection are stored. Mathematical sequences, sets, and bags are examples of possible mathematical models. Because there is no reason to preclude the collection from having entries that have equal priority, sets are not suitable for modeling the collection. Strings or sequences and bags (multisets) that allow duplicates fit the requirements better. Strings and sequences are typically useful to keep track of the arrival order of entries and allow ordering entries with the same priority on the basis of their arrival. Bags are better suited for modeling ob-

```
template <class Entry, class Entry_Compare_Capability, int Max_Size>
class Prioritizer_Template_2: public
Prioritizer_Template <Entry, Entry_Compare_Capability, Max_Size>
{
public:

/* A prioritizer object has three parts: an array ‘contents’, an Integer
‘size’, and a Boolean flag ‘filling’. The elements of the object are
stored in the contents from locations 0 to size - 1.

If the filling flag is false, then a heap exists between locations 0 and
size - 1, based on the ordering in class Item_Compare_Capability. The
smallest item is in location size - 1. */

  Prioritizer_Template_2 ();
/* Initializes size to 0 and filling to true. */

  virtual ~Prioritizer_Template_2 ();
  virtual void operator &= (Prioritizer_Template_1& rhs);

  virtual void Insert (Entry& x);
/* size is incremented and the new item is inserted into the array at
location size. */

  virtual void Change_Phase ();
/* Creates a heap of the array contents between locations 0 and size -
1, if the filling flag is true. */

  virtual void Remove_Next (Entry& x);
/* Returns the smallest element of array from location 0, and readjusts
the heap. */

  virtual void Remove_Any (Entry& x);
/* Returns the element of array from location size - 1 */

/* Rest of the public part and all of the private part is the same as in
Figure 4, except that Prioritizer_Template_2 is used instead of
Prioritizer_Template_1. */
...
};
```

Figure 6. Alternative implementation-based explanation of Prioritizer_Template.

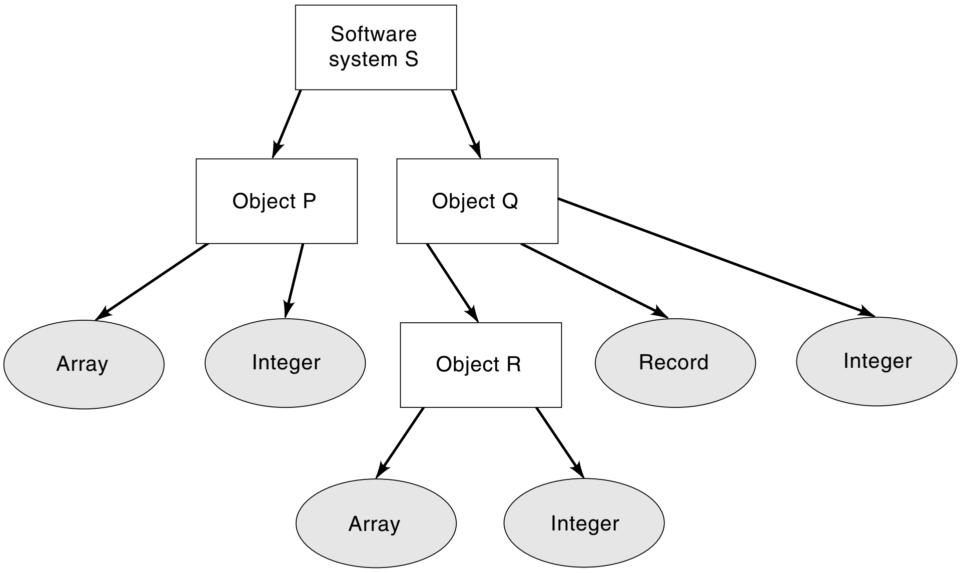

Figure 7. Object-based design without abstraction.

jects that disregard the order of arrival in prioritization, and these are the objects discussed in this paper.

Figure 8 contains a mathematical and implementation-independent specification of Prioritizer_Template in a variant of the RESOLVE notation (21). This concept includes the reusability design considerations discussed in the previous sections. Though this abstract description can be presented informally, the RESOLVE notation leads to a precise and understandable description also suitable for verifying correctness of its implementations.

The **concept** in Fig. 8 is parameterized by the type of entries to be placed in the prioritizer, the maximum size, and the ordering to be used in prioritization. The **restriction** states that the ordering must be a total preordering. For example, when Plane_Info is used as Entry type, ≤ operator on available fuel quantity is a suitable ordering for prioritizing planes for landing.

This concept **uses** typically "built-in" Boolean and Integer objects provided through concepts Standard_Boolean_Facility and Standard_Integer_Facility, respectively. Standard_Integer_Facility is a concept that explicitly models Integer objects with mathematical **integer** and Integer operations with mathematical integral operators. In the Standard_Boolean_Facility, Boolean values are used to model type Boolean. In general, description of a new concept might use a variety of other available concepts.

The **interface** section of the concept describes the type of objects and operations provided to users of the concept. In RESOLVE, the family of programming objects provided is described by one or a combination of mathematical models. Here, objects of **type family** Prioritizer are abstractly described as a mathematical pair: entry_count, a bag and insertion_phase, a **boolean.** Because notations for bags are not built into RESOLVE specification notation, a mathematical **subtype** BAG_MODEL is defined locally in this concept as a **function from** Entry **to** Integer. The definition TOTAL_ENTRY_COUNT_OF is a count of the number of entries in a bag. The local predicate IS_ONLY_DIFFERENCE is true if and only if its two bag arguments b1 and b2 differ only in that the first one has one more Entry x than the second one. BAG_MODEL and the two definitions are merely shorthand notations in the interface and are not needed otherwise.

A user can declare any number of objects of the type Prioritizer. Now suppose that p is an example of a Prioritizer. The concept specifies that its size is always within Max_Size, through the **constraints** assertion. Initially, as stated in the **initialization ensures** clause, there are no entries in the object, and it is in the insertion phase.

The rest of the interface section describes basic, permissible operations on Prioritizer objects. Operation Insert allows a new entry x to be added to a prioritizer p. It **alters** p, and **consumes** x: After a call to the operation, x has a legal value of type Entry, but it is not guaranteed to be what it was before the call or any other specific value. The operation requires that the prioritizer be in the insertion phase and that it not be full. The client of the object is responsible for calling the operation only when the **requires** clause is satisfied. It **ensures** that the only change to the prioritizer is that the count for the added entry increases by one. In the ensures clause, #p and #x denote the values of parameters 'p' and 'x' that are input to the operation; p and x denote their values after the operation.

Before clients remove items from a prioritizer, they must call the Change_Phase operation, which toggles the phase of the object. Operation Remove_Next removes and **produce**s the next "smallest" entry, based on the parametric ordering R. It *requires* that the machine *not* be in the insertion phase. In other words, though a client may interleave Insert and Remove_Next operations (as would be required in applications in class III discussed in this section), the Change_Phase operation must be called in between such interleaving. Otherwise, the client code is incorrect because then it violates the required condition of the Remove_Next operation.

Remove_Any operation is called when in either phase, and it returns an arbitrary item from the prioritizer. Based on the specification of Remove_Any, the code in Fig. 5 is incorrect (though it happens to produce correct results when Prioritizer_Template_1 implementation is used).

Operation Total_Entry_Count_Of **preserves,** that is, does not change the prioritizer for which the count is needed. This and Is_In_Insertion_Phase operations enable clients to check the required conditions on other operations. In the RESOLVE object design discipline, such "observer" operations to help check required conditions of other operations must be included in concepts for functional completeness. Operation Clear resets the prioritizer to its initial state. In addition to the operations explicitly specified in the concept, a user can also swap two Prioritizer objects. Notice that the Swap operator has been included in the C++ object design.

The set of basic primary operations is sufficiently powerful to manipulate Prioritizer objects effectively. Other *secondary* operations are constructed layered by using the basic operations. It is essential to keep the basic operations to a minimum to enhance understandability and, hence, reusability.

Understanding and Verification Using Abstract Descriptions. Assuming that implementations of Prioritizer and Queue objects satisfy their corresponding abstract descriptions, understanding and reasoning of code using these objects can be based on their abstract explanations as shown in Fig. 9. In an abstract explanation of Queue_Template, a queue is viewed mathematically as a **string of** Entries (24). In the figure, definitions IS_PERMUTATION and IS_ORDERED for mathematical strings have straightforward meanings and have been omitted. They are defined in the same way as TOTAL_ENTRY_COUNT_OF, defined in the Prioritizer_Template concept. The definition CONTENTS is a bag of entries that contain the entries in its parametric string and UNION is used to denote the union of two bags of entries.

To ease understanding and verifying the Queue Sort procedure, *loop invariants* have been provided through **maintaining** clauses. The **decreasing** clauses help show that the loops terminate. It is important to notice that understanding and verifying this code is independent of the actual implementation details. Regardless of which Prioritizer is used, the same understanding and reasoning hold.

The implementations of Prioritizer_Template_1 and Prioritizer_Template_2.2 are both correct with respect to the concept in Fig. 7. Though Prioritizer_Template_2 heapifies an array in the Change_Phase operation instead of just toggling its Boolean flag, it is still correct because, from a client's perspective, each operation produces intended effects (12).

```
concept Prioritizer_Template (
      type Entry
      constant Max_Size: Integer
      definition R (x, y: Entry): boolean
   )
   restriction Max_Size > 0 and
      for all x: Entry, R(x, x) and
      for all x, y, z: Entry, if R(x, y) and R(y, z) then R(x, z) and
      for all x, y: R(x, y) or R(y, x)

   uses Standard_Boolean_Facility, Standard_Integer_Facility

   math subtype BAG_MODEL is function from Entry to natural
   definition TOTAL_ENTRY_COUNT_OF(b: BAG_MODEL): integer =
                                              Σ b (x)
                                              x: Entry
   definition IS_ONLY_DIFFERENCE (x: Entry
                        b1, b2: BAG_MODEL): boolean =
              b2(x) = b1(x) + 1 and
              for all y: Entry, if y/= x then b1(y) = b2(y)
interface
      type family Prioritizer is modeled by (
         entry_count: BAG_MODEL
         insertion_phase: boolean
      )
         exemplar    p
         constraints TOTAL_ENTRY_COUNT_OF(p.entry_count) <= Max_Size
         initialization
            ensures  TOTAL_ENTRY_COUNT_OF(p.entry_count) = 0 and
                     p.insertion_phase = true

      operation Insert (
            alters p: Prioritizer
            consumes  x: Entry
         )
      requires    TOTAL_ENTRY_COUNT_OF(p.entry_count) < Max_Size and
                  p.insertion_phase = true
      ensures     IS_ONLY_DIFFERENCE(x, p.entry_count, #p.entry_count)
                  and p.insertion_phase = true
      operation Change_Phase (
            alters p: Prioritizer
         )
      ensures  p.entry_count = #p.entry_count and
               p.insertion_phase = note #p.insertion_phase

      operation Remove_Next (
            alters   p: Prioritizer
            produces x: Entry
         )
      requires    TOTAL_ENTRY_COUNT_OF(p.entry_count) > 0 and
                  p.insertion_phase = false
      ensures      for all y: Entry,
                      if p.entry_count(y) > 0 then R(x, y) and
                   IS_ONLY_DIFFERENCE (x, #p, p) and
                   p.insertion_phase = false

      operation Remove_Any (
            alters   p: Prioritizer
            produces x: Entry
         )
      requires    TOTAL_ENTRY_COUNT_OF(p.entry_count) > 0
      ensures     IS_ONLY_DIFFERENCE (x, #p, p) and
                  p.insertion_phase = #p.insertion_phase

      operation Is_In_Insertion_Phase (
            preserves p: Prioritizer
         ): Boolean
      ensures Is_In_Insertion_Phase = p.insertion_phase

      operation Total_Entry_Count_Of (
            preserves p: Prioritizer
         ): Integer
      ensures  Total_Entry_Count_Of = TOTAL_ENTRY_COUNT_OF(p.entry_count)

      operation Allowed_Max_Size (
         ): Integer
      ensures  Allowed_Max_Size = Max_Size

      operation Clear (
            alters    p: Prioritizer
         )
      ensures     TOTAL_ENTRY_COUNT_OF(p.entry_count) = 0 and
                  p.insertion_phase = true

end Prioritizer_Template
```

Figure 8. A conceptualization of prioritizer template.

```
procedure Queue_Sort (
      alters    q: Queue
   )
ensures     IS_PERMUTATION(#q, q) and IS_ORDERED(q)
   p: Prioritizer
   x: Entry
begin
   maintaining union (p.entry_count, contents(q)) = contents (#q)
               and p.insertion_phase
   decreasing |q|
   while Length_Of(q) > 0 loop
      Dequeue(q, x)
      Insert(p, x)
   end loop
   Change_Phase(p)
   maintaining union(p.entry_count, contents(q)) = contents (#q)
               and is_ordered (q) and not p.insertion_phase
   decreasing |TOTAL_ENTRY_COUNT_OF(p.entry_count)|
   while Total_Entry_Count_Of(p) > 0 loop
      Remove_Next(p, x)
      Enqueue(q, x)
   end loop
end Queue_Sort
```

Figure 9. The role of conceptualizations in understanding and reasoning.

Remove_Next, for example, returns the smallest item from the collection of inserted items as demanded by the specification. Modular verification and testing of parameterized objects and objects arising from recasting optimization algorithms are fundamental areas of reuse research (12,15,25).

Discussion on Notations for Conceptualization. Modern programming languages, such as Ada, C++, and Java, include features to facilitate construction of object-based software components through separation of public interfaces and private implementations, inheritance, and parameterization mechanisms. The interfaces must be explained abstractly to reap significant benefits of reuse. Detailed implementation-oriented comments cannot replace the need for suitable abstraction.

Though abstract descriptions can be written in formal notations or in natural languages, such as English, formal explanations are most suitable for reusable software components because they are precise and they facilitate human communication without requiring any common understanding, except standard mathematical symbols. Though we have used the RESOLVE notation in this paper, other formal specification languages, such as Larch, VDM, and Z, are also appropriate (26).

Use of a formula notation alone, however, does not guarantee a good conceptualization. For widespread reuse, the concepts must be suitably generalized and must permit several plug-compatible implementations to provide performance tradeoffs. In addition, if unsuitable mathematical models are used in conceptualizing a problem, understandability might be compromised (24,27). Because reusable concepts are likely to be read far more often than they are written, understandability is a fundamental requirement. The RESOLVE notation used in this article is arguably quite appropriate because it has been regularly used for instruction in classrooms at the freshman level.

OTHER TECHNICAL AND NONTECHNICAL REUSE CONSIDERATIONS

Identification of new reusable concepts that provide higher level building blocks and raise the level of software construction is a challenging activity and involves considerable research. Earlier work on reuse focused on subroutine libraries for languages, such as FORTRAN, and development of object-based components that capture traditional data structures, such as stacks, queues, and lists (28). More recently, by recasting classical algorithms (12), a variety of previously unrecognized reusable concepts, such as the Prioritizer_ Template have been discovered. Other examples of recasting include Minimum_Spanning_Forest_Template and Cheapest_ Path_Template, where graph optimization algorithms are recast as reusable concepts.

In addition to specific reusable concepts, metalevel concepts that capture common interface models and enable easier understanding of a wide variety of other concepts are receiving attention (13). Identification and description of commonly used patterns, styles, and architectures in software systems are among active areas of reuse research (29,30). A key objective of this research is to minimize the cost of understanding during software evolution and reuse. To facilitate ease of sharing and integration of preexisting components, industry standards, such as CORBA and COM/DCOM (31), have been proposed. Whereas the focus in such work is on general-purpose descriptions and integration, the objectives of research in domain analysis and engineering are in identifying concepts commonly used within an application domain (32). In current practice, domain-specific artifacts and architectures are typically described with informal domain terminology shared by the intended audience. Reusing components within restricted domains, researchers have shown that it is possible to generate software systems effectively (33). Abstraction and conceptualization have the potential for widening the scope of domains and enhancing the applicability of reusable artifacts.

Both general-purpose and domain-specific artifacts need to be classified, stored, and retrieved through reusable software repositories or libraries. Recent studies have shown that keyword-based search is quite effective and adequate for artifact retrieval (34). The difficult challenge is ensuring that the artifacts in the library are of high quality and are well-designed to facilitate ease of reuse. Although reusing a *legacy* software system or component not designed to be reused may have little potential for significant benefits because of the overriding costs in understanding and reasoning, legacy software can prove to be a valuable source for identifying new reusable concepts within and across domains. The general problem of reverse engineering a poorly designed software system to a well-designed one, however, is arguably intractable in the usual computational complexity sense (35).

Although several technical obstacles for exploiting the full potential of software reuse still remain to be tackled, significant progress has been made in introducing a software reuse process into organizations (3,7,36). Successful software reuse requires considerable up-front investment in building suitably generalized and conceptualized reusable artifacts. Though the cost is amortized over the long run through many reuses, managerial reluctance to make the initial investment for potential long-term gains remains to be overcome. To highlight the long-term benefits of reuse, measurement techniques and reuse cost-benefit analyses have been developed, and empirical studies have been conducted (36,37). Other than questions of economics, legal questions must also be addressed because the success of software reuse hinges upon developers marketing their products in a way that does not compromise their ownership rights.

SUMMARY

Any successful software engineering process must include considerations of reusability. Although reuse is an essential mechanism for improving both software productivity and quality, reuse does not imply automatically that significant improvements result. Benefits result from reusing a software artifact only in direct proportion to the effort invested in conceptualizing and developing a high-quality artifact.

Software reuse raises both challenges and opportunities. It is easier to justify the investment in analysis, conceptualization, implementation, and verification of reusable software systems and components because the cost in these efforts are amortized over the many uses and in their evolution. Reusable software design must include simultaneous considerations of a number of factors including generalization and abstraction and must take advantage of modern specification and programming language mechanisms such as objects, inheritance, and parameterization. The potential economic benefits of well-designed artifacts need to be demonstrated by convincing evidence to encourage managers to invest in reusable software construction.

ACKNOWLEDGMENTS

I would like to thank members of the Reusable Software Research Groups at The Ohio State University and at West Virginia University for discussions on various aspects of this work. In particular, my thanks are due to Bruce Weide with whom I have had several conversations regarding the specific contents of this paper. I gratefully acknowledge the financial support for this research from:

- *The National Science Foundation* under grant CCR-9204461.
- *The Advanced Research Projects Agency of the Department of Defense* under ARPA contract number DAAH04-94-G-0002 and DAAH04-96-1-D419, both monitored by the U.S. Army Research Office.
- *The National Aeronautics and Space Administration* under grant NCC 2-979, administered by Software Independent Verification and Validation Facility through Ames Research Center.

BIBLIOGRAPHY

1. W. Gibbs, Software's Chronic Crisis, *Sci. Amer.,* 86–95, 1994.
2. T. J. Biggerstaff and A. J. Perlis (eds.), *Software Reusability,* New York: ACM Press, 1989, Vols. 1 and 2.
3. W. Frakes and S. Isoda (eds.), Special issue on systematic reuse, *IEEE Software,* **11** (5): 1994.
4. M. Harandi (ed.), *Proc. 1997 Symp. on Software Reusability,* New York: ACM Press, 1997.
5. L. Latour (ed.), *Proc. Annual Workshops on Software Reuse,* Online proceedings at http://www.umcs.maine.edu.
6. A. Mili, H. Mili, and F. Mili, Reusing software: Issues and research directions, *IEEE Trans. Soft. Eng.,* **21**: 1995.
7. M. Sitaraman (ed.), *Proc. 4th Int. Conf. on Software Reuse,* Los Alamitos: IEEE Computer Society Press, 1996.
8. B. W. Weide and J. E. Hollingsworth, Scalability of Reuse Technology to Large Systems Requires Local Certifiability, *Proc. 5th Annual Workshop on Software Reuse,* Palo Alto, CA, 1992.
9. J. E. Hopkins and M. Sitaraman, Software Quality Is Inversely Proportional to Potential Local Verification Effort, *Proc. Sixth Annual Workshop on Software Reuse,* Owego, NY, 1993.
10. J. A. Goguen, Principles of Parameterized Programming, in T. J. Biggerstaff and A. J. Perlis, *Software Reusability, Concepts and Models,* Reading, MA: Addison–Wesley, 1989, Vol. 1.
11. B. Meyer, *Reusable Software: The Base Object-Oriented Component Libraries,* Prentice-Hall International, 1994.
12. B. W. Weide, W. F. Ogden, and M. Sitaraman, Recasting algorithms to encourage reuse, *IEEE Software,* **11** (5): 80–88, 1994.
13. S. Edwards and L. Latour, The need for good mental models of software subsystems—Working group report, *Proc. Seventh Annual Workshop on Software Reuse,* St. Charles, IL, 1995.
14. W. Frakes, L. Latour, and T. Wheeler, Descriptive and prescriptive aspects of the 3C model, *Proc. Third Annual Workshop on Software Reuse,* Syracuse, NY, 1990.
15. G. W. Ernst et al., Modular verification of Ada generics, *Computer Languages,* **16** (3/4), 259–280, 1991.
16. G. Leavens, Modular specification and verification of object-oriented programs, *IEEE Software,* **8** (4): 72–80, 1991.
17. W. F. Ogden et al., The RESOLVE framework and discipline, *ACM SIGSOFT Software Engineering Notes,* **19** (4): 25–37, 1994.
18. S. H. Zweben et al., The effects of layering and encapsulation on software development cost and quality, *IEEE Trans. Soft. Eng.,* **21**: 1994.
19. D. Fleming, M. Sitaraman, and S. Sreerama, A practical performance criterion for object interface design, *Journal of Object-Oriented Programming,* New York: SIGS Publication, 1997.
20. D. E. Harms and B. W. Weide, Copying and swapping: Influences on the design of reusable software components, *IEEE Trans. Soft. Eng.,* **17**: 424–435, 1991.
21. J. Hollingsworth et al., RESOLVE components in Ada and C++, *ACM SIGSOFT Soft. Eng. Notes,* **19** (4): 52–63, 1994.
22. M. Sitaraman, A class of programming language mechanisms to facilitate multiple implementations of a specification, *Proc. International Conference on Computer Languages,* Los Alamitos: IEEE Computer Society Press, 1992, pp. 182–191.
23. S. Sreerama, D. Fleming, and M. Sitaraman, Graceful object-based performance evolution, *Software—Practice & Experience,* 1997.

24. M. Sitaraman, L. R. Welch, and D. E. Harms, On specification of reusable software components, *Int. J. Software Eng. Knowledge Eng.,* **3** (2): 207–219, 1993.

25. S. H. Zweben, W. D. Heym, and J. Kimmich, Systematic testing of data abstractions based on software specifications, *J. Software Testing, Verification, and Reliability,* **1** (4): 39–55, 1992.

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

27. B. W. Weide et al., Characterizing observability and controllability of software components, *Proc. Fourth Int. Conf. on Software Reuse,* Los Alamitos: IEEE Computer Society Press, 1996, pp. 62–71.

28. G. Booch, *Software Components with Ada: Structures, Tools, and Subsystems,* Menlo Park, CA: Benjamin–Cummings, 1997.

29. D. Garlan and D. E. Perry (eds.), Introduction to the special issue on software architecture, *IEEE Trans. Soft. Eng.,* **21**: 269–274, 1995.

30. S. J. Mellor and R. Johnson (eds.), Object methods, special issue on patterns, and architectures, *IEEE Software,* **14** (1): 1997.

31. V. Kozaczynski and J. Q. Ning, (Moderators), Panel on component-based software engineering, *Proc. Fourth Int. Conf. on Software Reuse,* Los Alamitos: IEEE Computer Society Press, 1996, pp. 236–241.

32. R. Prieto-Diaz and G. Arango (eds.), *Domain Analysis and Software Systems Modeling,* Los Alamitos: IEEE Computer Society Press, 1991.

33. D. Batory and B. J. Geraci, Composition validation and subjectivity in GenVoca generators, *IEEE Trans. Soft. Eng.,* **23**: 67–82, 1997.

34. W. B. Frakes and T. Pole, An empirical study of representation methods for reusable software components, *IEEE Trans. Soft. Eng.,* **20**: 617–630, 1994.

35. B. W. Weide, J. Hollingsworth, and W. Heym, Reverse engineering of legacy code exposed, *Proc. 17th Int. Conf. on Soft. Eng.,* New York: ACM, 1995, pp. 327–331.

36. W. Tracz, *Confessions of a Used-Program Salesman,* Reading, MA: Addison–Wesley, 1995.

37. J. Poulin, *Measuring Software Reuse: Principles, Practices, and Economic Models,* Reading, MA: Addison–Wesley, 1997.

Murali Sitaraman
West Virginia University