Common Interface Models for Components Are Necessary to Support Composability

Stephen H. Edwards

Department of Computer and Information Science
The Ohio State University
2036 Neil Avenue Mall
Columbus, OH 43210

Abstract

Unfortunately, reusable components are often based on different conceptual models of behavior. The model underlying the specification of one module's parameter requirements may significantly differ from the model underlying the specification of another module's exported features, even if the two modules intuitively seem compatible. There is no well understood groundwork of common models for component interaction, and the lack of guidelines for applying these models exacerbate the composability problem. This paper will describe how varying interface models and techniques for describing a component's interface requirements affect composability. These problems will be illustrated in the context of common interface properties that are exhibited even in simple components.

Keywords: component reuse, module composition, intensional and extensional interfaces, modeling interface behavior.

1 Introduction

As more and more attention is being focused on software reuse, new problems are emerging. While the majority of these difficulties are rooted in nontechnical issues, it is clear that there are more technical problems than previously anticipated. One of these technically oriented problems arises out of the constructive, or parts based, approach to software reuse[1]—combining software components as basic building blocks to form larger structures.

Specifically, in practice it can be very difficult to "compose" or interlock two apparently general software components. Of course, there are many factors affecting the composability of software

[1]The distinction between the constructive, composition based approach and the generative approach to reuse is described in [BR87].

components. This paper will concentrate on one of these factors: components are often based on different conceptual models of behavior.

There is no well understood groundwork of common models for component interaction, and the lack of guidelines for applying these models exacerbate the composability problem. This paper will describe how interface models affect composability and how techniques for describing a component's interface requirements affect composability. These problems will be illustrated in the context of common interface properties that are exhibited even in simple abstract data type ( ) components.

2 Composing Software Components

Before tackling the problems that arise when trying to compose software components, it is necessary to define what "composition" really is. In this paper, composition means binding two components together along a common interface boundary. Further, one of the components *exports* while the other *imports*—the exporter provides facilities that are needed by the importer. This definition is directly derived from the work of Goguen [Gog84, Gog86].

The concept of composing software components is central to the "parts oriented" reuse approach. Ideally, a module has a well-defined description of the interface it provides to its clients. Also, a module ideally has a well-defined set of interface requirements that completely delineate what must be supplied to it during composition. Thus, the interface of the exporter and the requirements of the importer can be compared to check for valid compositions.

By examining these ideas in more detail, one can see that a component has a set of requirements that must be fulfilled by the other components to which it is connected. It is best if these requirements, or "interface expectations," are explicitly defined, although in some languages they may be left implicit. In Ada, for example, interface expectations for module composition can be (syntactically) defined in the form of generic parameters to the module. Any other module that exports types and operations consistent with these requirements (i.e., that conform to the generic parameters) can be composed with such a component. The way these distinct components fit together via composition is determined by the facilities exported by one, and the parameter expectations of the other (e.g., generic parameters).

The concept of module composition can be further refined by considering the purpose of combining modules. Specifically, components can be composed both *horizontally* and *vertically* [Gog84]. Horizontal composition is the interconnection of components at a single level of abstraction so that they may work together to form a cohesive abstract layer. Vertical composition is the construction of higher levels of abstraction on top of existing layers.

To illustrate the concept of horizontal composition, consider two Ada generic packages, one that exports a queue type and one that exports a list type. If one instantiated the queue generic using the list type, a "queue of lists" facility would be created. This combines the two components to form a larger abstract machine, and is thus horizontal composition. On the other hand, Consider implementing the queue package so that its body uses an instantiation of the list package for representing actual queues. This involves building a new abstract layer on top of existing layers, and is thus vertical composition.

Irrespective of the type of composition, however, each component has a set of (possibly implicit) requirements that must be met when it is composed with other components. It is the nature of

these interface expectations, and how they might be met by other potentially compatible modules that is the subject of the remainder of this paper.

3 Interface Models

In simple terms, an "interface model" is just an abstraction of a set of types and operations, and how they are used together. For example, one can talk about the interface model imported by a component when discussing its interface requirements. This refers to the conceptual idea of how the imported facilities (types and operations) interact with each other. Likewise, the interface model exported by a module refers to the abstract concept that the module embodies, and how each of the exported operations interacts with and affect that abstraction.

The critical element in understanding this idea of an interface model lies in the fact that it is *more* than simply a collection of types and operations. It also encompasses how the individual types and operations interact, and how they are used in conjunction with each other to achieve certain goals. The underlying behavior specified in a description of the interface is crucial.

Given this idea, it is easy to see that models of interface behavior can conflict if the interface of one component is designed around one model of interaction, while the expectations of another component are designed around a different model. It may be difficult to compose the two, that is, to use the first component to supply the needed facilities of the second.

To see how interface models can conflict, consider the specification of the function `Find_the_Max_of` presented in Figure 1, which is taken from [Tra89]. This generic function, which is a generalization of an Ada `for` loop over an array structure, is parameterized with its interface expectations. In order to compose some with this function, that must export the required operations. If there is not an exact match between the importer's requirements and the exporters facilities, then "glue" code may be written that implements the required operations in terms of those provided by the exporter.

For this example, it is easy to see how appropriate glue for an Ada array type can be written to satisfy `Find_the_Max_of`'s requirements. However, this function's requirements are based upon an interface model that is perhaps too tightly tied to the concept of an array. In general, one might expect a maximum-finding routine to operate over a set (or multiset) of values. Intuitively, the only requirement is that there be a mechanism for iterating over the collection of values.

To illustrate this, imagine how one might use `Find_the_Max_of` on a collection of elements represented as a list. Depending on the interface model provided by the list abstraction, the cost of providing the necessary glue could be prohibitive, both in terms of the cost of implementing it and its efficiency. Consider the list abstraction presented in Figure 2, which represents one of many alternative ways of specifying the basic operations necessary to access a list structure[2].

Because the list abstraction presented by the `One_Way_List` package in Figure 2 restricts clients to accessing list elements in sequential order, there is a clear mismatch between its exported functionality and the import requirements of `Find_the_Max_of`. Similarly, Figure 3 presents another data abstraction, a variable length sequence[3], which is built on yet another behavioral model.

[2]This specification is derived from the work of the Reusable Software Research Group at the Ohio State University. See [WOZ91] for a description of the rationale behind similar designs.

[3]This specification is also derived from the work of the Reusable Software Research Group at the Ohio State

Figure 1: The Example `Find_the_Max_of` Generic Function

```
generic
    type Element is limited private;
    type Index is limited private;
    type Vector is limited private;
    with function "<"(Left, Right: in Element) return Boolean is <>;
    with function "="(Left, Right: in Index) return Boolean = <>;
    with function Is_Empty(The_Vector: in Vector) return Boolean;
    with function First_Index_of(The_Vector: in Vector) return Index;
    with function Last_Index_of(The_Vector: in Vector) return Index;
    with function Next_Index(Previous: in Index) return Index;
    with function Get_Element(From       : in Vector;
                              At_Location: in Index) return Element;
    with procedure Assign(Into: in out Element;
                          From: in Element);
    with procedure Assign(Into: in out Index;
                          From: in Index);
function Find_the_Max_of(The_Vector: in Vector) return Element;
```

In this case, access to elements of a sequence can be specified by position. However, individual items within a sequence are accessed through the use of a `Swap_Entry` procedure that exchanges the object within the sequence with one provided by the caller. This is clearly different from the value preserving nature of `Find_the_Max_of`'s `Get_Element` function.

In both of these cases, it is possible to write glue routines to wedge the given abstractions to fit the interface requirements of `Find_the_Max_of`, given enough time. However, the cost associated with writing and maintaining this glue reduces the benefits that one obtains from reusing the `Find_the_Max_of` component in the first place. Further, in more complex cases, the difficulty of composing two modules may likely result lead one to forgo composition and write a "compatible" module from scratch.

To most experienced designers, especially if aided by hindsight, it is clear that these costs may be avoided simply by redesigning the interface requirements of the given unit. If one looks at both the exporter and importer at the same time, it is a much easier exercise to write interfaces based on compatible models of interaction. In general, however, the importer is written independently of the exporter. One possible way to avoid this difficulty is to attempt to evolve interface models for frequently used behavioral models that can eventually become commonly accepted *de facto* interface standards.

For example, the primary interface requirement for the `Find_the_Max_of` function is the existence of some collection of values over which one can iterate. If a common model for the set

University.

Figure 2: One Possible List Abstraction in Ada

```
generic
    type Item is limited private;
    ...
package One_Way_List is

    type List is limited private;
    procedure Initialize(The_List: in out List);
        -- Initially, a list is empty, and the current position in the
        -- list is at the beginning.
    procedure Finalize(The_List: in out List);
        -- This frees up resources bound to The_List.
    procedure Swap(Left, Right: in out List);
        -- Exchanges two lists.

    procedure Add_Right(To_the_List: in out List;
                        New_Entry  : in out Item);
        -- Places a new object to the right of the current position in
        -- the specified list.  New_Entry is set to an initialized value.

    procedure Remove_Right(From_the_List: in out List;
                           The_Entry    : in out Item);
        -- Removes the object immediately to the right of the current
        -- position in the specified list.  The old value originally
        -- in The_Entry is finalized, then the removed object is returned
        -- as The_Entry.

    procedure Advance(The_List: in out List);
        -- Advances the current position in The_List one object to the
        -- right.

    function At_Left_End(of_the_List: in List) returns boolean;
        -- ...
    function At_Right_End(of_the_List: in List) returns boolean;
        -- ...
    procedure Rewind_Left(The_List: in out List);
        -- ...

end One_Way_List;
```

Figure 3: One Possible Sequence Abstraction in Ada

```
generic
    type Item is limited private;
    ...
package Variable_Length_Sequence is

    type Sequence is limited private;
    procedure Initialize(The_List: in out List);
    procedure Finalize(The_List: in out List);
    procedure Swap(Left, Right: in out List);

    procedure Insert(Into_S   : in out Sequence;
                     At_Pos   : in     Integer;
                     New_Entry: in     Item);
        -- Adds a new element to the given sequence at a given position,
        -- moving all elements from At_Pos to the end of the sequence to
        -- the right one position.  Positions are numbered from 0, and
        -- giving an At_Pos value equal to the length of a sequence will
        -- append an new element to the end.

    procedure Remove(From_S   : in out Sequence;
                     At_Pos   : in     Integer;
                     The_Entry: in out Item);
        -- Removes the given element from the specified sequence, moving
        -- all subsequent elements up one location.  The original value
        -- held in The_Entry is finalized, then the removed element is
        -- placed in The_Entry.

    function Size_of(S: in Sequence) returns Integer;
        -- ...

    procedure Swap_Entry(In_S  : in out Sequence;
                         At_Pos: in     Integer;
                         Entry : in out Item);
        -- Exchanges Entry with the value at the specified position in S.

end Variable_Length_Sequence;
```

of operations that form an iteration capability can be adopted[4], then the generic parameters of `Find_the_Max_of` can be expressed using this form. Likewise, container structures can adopt the same form when providing operations for clients to use in constructing iterations. Other areas that are prime candidates for interface model exploration include the concurrency protection scheme adopted in a component, the memory management approach used by a component, the approach to file I/O for an , and so on.

Unfortunately, designing "consistent" component interfaces is extremely difficult. Few general guidelines on commonly used interface models are available, and how these models affect composability isn't understood. In fact, simply identifying the conceptual model behind a given component can be hard. Just defining the concurrency protection approach used in a given component is often difficult, for example. Such aspects of a component's interface, such as the concurrency protection scheme, the exported iterator structures, the memory management approach, and so on, are often crafted for a specific implementation. As a result, "common" models for those aspects of a component's interface aren't commonly applied!

As a result, even if the abstraction supplied by one component is conceptually the same as that required by another, there is a good possibility there will be a difference in the actual interfaces. In general, this problem is most likely unsolvable, but it is possible to work towards standards that make interoperability of component interfaces more practical, common, and cheaper.

4 Extensional Versus Intentional Interfaces

Another issue that can affect the compatibility of two components is the manner in which their interface requirements are expressed. There is a difference between *extensionally* and *intensionally* expressing interface requirements [Lat89]. Items that satisfy extensional requirements are typically defined by inclusion in a specified (possibly infinite) set of items. Items that satisfy intensional requirements, on the other hand, are defined by characteristic properties they must possess.

One example of an extensional interface requirement often appears in object-oriented languages: parameters must belong to a specific class. While classes can usually be extended via specialization, the conformance to an interface specification is determined by name equivalence to the parameter's class.

An example of an intensional interface requirement is a generic parameter to an Ada package. Here, conformance to the interface specification is determined by *structural* equivalence. For example, if a `private` type parameter is required, any Ada type that has the same properties as a `private` type—the presence of built-in equality and assignment operators, and so on—will conform to the interface.

More flexible intensional interface mechanisms are present in [Gog84, Gog86, Tra90]. supports formally defined correspondences between packages. As a result, when packages are used as generic parameters, any other package that can be shown to provide a conforming set of operations and types can be used to satisfy the interface requirements. With the addition of formal semantic definitions as in [Tra90], the structural equivalence can be extended to semantic equivalence.

[4]See [Edw90] for one recommendation of a standard iterator profile.

Clearly extensional requirements are more limiting than intensional ones. The correspondence facilities present in languages like allow one the freedom to express interface requirements in one way, but still meet those requirements with any component that provides the correct abstraction. While the restrictiveness of extensional interface requirements in an object-oriented programming language initially seems inconsequential, relieving it can create more problems. The use of multiple inheritance can arbitrarily broaden extensional requirements; however, using a single inheritance hierarchy for many purposes is fraught with difficulty [LaL89].

Also, generic parameters can be extensional. The programming language allows packages as generic parameters, but uses name equivalence rather than structural equivalence to determine conformance [Heg89]. As a result, a mechanism that on the surface appears very much like Ada's generic mechanism actually gives rise to extensional rather than intensional behavior.

The restrictive nature of extensional requirements specifications simply inflames the composability problem. It further restricts, or even overspecifies, the interface requirements so that even components with potentially compatible models of behavior cannot be combined.

5 Conclusion

Common models of interface behavior are necessary for promoting composability in a software component industry. Further, defining canonical models and providing guidelines for their application is hard. Such models are not currently in use because most components use highly tailored, implementation specific behavioral models. Also, the method by which interface expectations are specified can further limit module compatibility. This limiting can arise from extensional interfaces that can prematurely restrict the modules that can be used to supply a given component's needs, while intensional interfaces seem to offer more generality. As a result, interface models and methods for expressing interface requirements should be explored in order to increase the likelihood that reusable modules are in fact composable with other reusable modules.

References

[BR87] Ted Biggerstaff and Charles Richter. Reusability framework, assessment, and directions. *IEEE Software*, 4(2):41–49, March 1987.

[Edw90] Stephen H. Edwards. An approach for constructing reusable software components in Ada. IDA Paper P-2378, Institute for Defense Analyses, Alexandria, VA, September 1990.

[Gog84] Joseph A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, September 1984.

[Gog86] Joseph A. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 19(2):16–28, February 1986.

[Heg89] Wael A. Hegazy. *The Requirements of Testing a Class of Reusable Software Components*. PhD thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, 1989.

[LaL89] Wilf R. LaLonde. Designing families of data types using exemplars. *ACM Transactions on Programming Languages and Systems*, 11(2):212–248, April 1989.

[Lat89] Larry Latour. University of Maine, personal communication, 1989.

[Tra89] Will Tracz. Parameterization: A case study. *Ada Letters*, IX(4):92–102, May/June 1989.

[Tra90] William Tracz. *Formal Specification of Parameterized Programs in LILEANNA*. PhD thesis, Dept. of Electrical Engineering, Stanford University, Stanford, CA, 1990.

[WOZ91] Bruce W. Weide, William F. Ogden, and Stuart H. Zweben. Reusable software components. In M. C. Yovits, editor, *Advances in Computers*. Academic Press, 1991.