SOFTWARE RECLAMATION:
Improving Post-Development Reusability

John W. Bailey and Victor R. Basili

The University of Maryland Department of Computer Science
College Park, Maryland 20742

Abstract

This paper describes part of a multi-year study of software reuse being performed at the University of Maryland. The part of the study which is reported here explores techniques for the transformation of Ada programs which preserve function but which result in program components that are more independent, and presumably therefore, more reusable. Goals for the larger study include a precise specification of the transformation technique and its application in a large development organization. Expected results of the larger study, which are partially covered here, are the identification of reuse promoters and inhibitors both in the problem space and in the solution space, the development of a set of metrics which can be applied to both developing and completed software to reveal the degree of reusability which can be expected of that software, and the development of guidelines for both developers and reviewers of software which can help assure that the developed software will be as reusable as desired.

The advantages of transforming existing software into reusable components, rather than creating reusable components as an independent activity, include: 1) software development organizations often have an archive of previous projects which can yield reusable components, 2) developers of ongoing projects do not need to adjust to new and possibly unproven methods in an attempt to develop reusable components, so no risk or development overhead is introduced, 3) transformation work can be accomplished in parallel with line developments but be separately funded (this is particularly applicable when software is being developed for an outside customer who may not be willing to sustain the additional costs and risks of developing reusable code), 4) the resulting components are guaranteed to be relevant to the application area, and 5) the cost is low and controllable.

Introduction

Broadly defined, software reuse includes more than the repeated use of particular code modules. Other life cycle products such as specifications or test plans can be reused, software development processes such as verification techniques or cost modeling methods are reusable, and even intangible products such as ideas and experience contribute to the total picture of reuse [1,2]. Although process and tool reuse is common practice, life cycle product reuse is still in its infancy. Ultimately, reuse of early lifecycle products might provide the largest payoff. For the near term, however, gains can be realized and further work can be guided by understanding how software can be developed with a minimum of newly-generated source lines of code.

The work covered in this paper includes a feasibility study and some examples of generalizing, by transforming, software source code after it has been initially developed, in order to improve its reusability. The term software reclamation has been chosen for this activity since it does not amount to the development of but rather to the distillation of existing software. (Reclamation is defined in the dictionary as obtaining something from used products or restoring something to usefulness [3].) By exploring the ability to modify and generalize existing software, characterizations of that software can be expressed which relate to its reusability, which in turn is related to its maintainability and portability. This study includes applying these generalizations to several small example programs, to medium sized programs from different organizations, and to several fairly large programs from a single organization.

Earlier work has examined the principle of software reclamation through generic extraction with small examples. This has revealed the various levels of difficulty which are associated with generalizing various kinds of Ada dependencies. For example, it is easier to generalize a dependency that exists on encapsulated data than on visible data, and it is easier to generalize a dependency on a visible array type than on a visible record type. Following that work, some medium-sized examples of existing software were analyzed for potential generalization. The limited success of these efforts revealed additional guidelines for development as well as limitations of the technique. Summaries of this preceding work appear in the following sections.

Used as data for the current research is Ada software from the NASA Goddard Space Flight Center which was written over the past three years to perform spacecraft simulations. Three programs, each on the order of 100,000 (editor) lines, were studied. Software code reuse at NASA/GSFC has been practiced for many years, originally with Fortran developments, and more recently with Ada. Since transitioning to Ada, management has observed a steadily increasing amount of software reuse. One goal which is introduced here but which will be addressed in more detail in the larger study is the understanding of the nature of the reuse being practiced there and to examine the reasons for the improvement seen with Ada. Another goal of this as well as the larger study is to compare the guidelines derived from the examination of how different programs yield to or resist generalization. Several questions

are considered through this comparison—including the universality of guidelines derived from a single program and whether the effect of the application domain, or problem space, on software reusability can be distinguished from the effect of the implementation, or solution space.

Superficially, therefore, this paper describes a technique for generalizing existing Ada software through the use of the generic feature. However, the success and practicality of this technique is greatly affected by the style of the software being transformed. The examination of what characterizations of software are correlated with transformability has led to the derivation of software development and review guidelines. It appears that most, if not all, of the guidelines suggested by this examination are consistent with good programming practices as suggested by other studies.

The Basic Technique

By studying the dependencies among software elements at the code level, a determination can be made of the reusability of those elements in other contexts. For example, if a component of a program uses or depends upon another component, then it would not normally be reusable in another program where that other component was not also present. On the other hand, a component of a software program which does not depend on any other software can be used, in theory at least, in any arbitrary context. This study concentrates only on the theoretical reusability of a component of software, which is defined here as the amount of dependence that exists between that component and other software components. Thus, it is concerned only with the syntax of reusable software. It does not directly address issues of practical reusability, such as whether a reusable component is useful enough to encourage other developers to reuse it instead of redeveloping its function. The goal of the process is to identify and extract the essential functionality from a program so that this extracted essence is not dependent on external declarations, information, or other knowledge. Transformations are needed to derive such components from existing software systems since inter-component dependencies arise naturally from the customary design decomposition and implementation processes used for software development.

Ideal examples of reusable software code components can be defined as those which have no dependencies on other software. Short of complete independence, any dependencies which do exist provide a way of quantifying the reusability of the components. In other words, the reusability of a component can be thought of as inversely proportional to the amount of external dependence required by that component. However, some or all of that dependence may be removable through transformation by generalizing the component. A measure of a component's dependence on its externals which quantifies the difficulty of removing that dependence through transformation and generalization is slightly different from simply measuring the dependence directly, and is more specifically appropriate to this study. The amount of such transformation constitutes a useful indication of the effort to reuse a body of software.

Both the transformation effort and the degree of success with performing the transforms can vary from one example to the next. The identification of guidelines for developers and reviewers was made possible by observing what promoted or impeded the transformations. These guidelines can also help in the selection of reusable or transformable parts from existing software. Since dependencies among software components can typically be determined from the software design, many of the guidelines apply to the design phase of the life cycle, allowing earlier analysis of reusability and enabling possible corrective action to be taken before a design is implemented. Although the guidelines are written with respect to the development and reuse of systems written in the Ada language, since Ada is the medium for this study, most apply in general to software development in any language.

One measure of the extent of the transformation required is the number of lines of code that need to be added, altered, or deleted [4]. However, some modifications require new constructs to be added to the software while others merely require syntactic adjustments that could be performed automatically. For this reason, a more accurate measure weighs the changes by their difficulty. A component can contain dependencies on externals that are so intractable that removing them would mean also removing all of the useful functionality of the component. Such transformations are not cost-effective. In these cases, either the component in question must be reused in conjunction with one or more of the components on which it depends, or it cannot be generalized into an independently reusable one. Therefore, for any given component, there is a possibility that it contains some dependencies on externals which can be eliminated through transformation and also a possibility that it contains some dependencies which cannot be eliminated.

To guide the transformations, a model is used which distinguishes between software function and the declarations on which that function is performed. In an object-oriented program (for here, a program which uses data abstraction), data declarations and associated functionality are grouped into the same component. This component itself becomes the declaration of another object. This means the function / declaration distinction can be thought of as occurring on multiple levels. The internal data declarations of an object can be distinguished from the construction and access operations supplied to external users of the object, and the object as a whole can be distinguished from its external use which applies additional function (possibly establishing yet another, higher level object). The distinction between functions and objects is more obvious where a program is not object-oriented since declarations are not grouped with their associated functionality, but rather are established globally within the program.

At each level, declarations are seen as application-specific while the functions performed on them are seen as the potentially generalizable and reusable parts of a program. This may appear backwards initially, since data abstractions composed of both declarations and functions are often seen as reusable components. However, for consistency here, functions and declarations within a data abstraction are viewed as separable in the same way as functions which depend on declarations contained in external components are separable from those declarations. In use, the reusable, independent functional components are composed with application-specific declarations to form objects, which can further be composed with other independent functional components to implement an even larger portion of the overall program.

Figure 1 shows one way of representing this. All the ovals are objects. The dark ones are primitives which have predefined operations, such as integer or Boolean. The white ovals represent program-supplied functionality which is composed with their contained objects to form a higher level

6109

object. The intent of the model is to distinguish this program-specific functionality and to attempt to represent it independently of the objects upon which it acts.

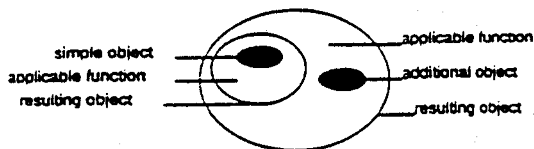

Figure 1.

Some Ada which might be represented as in the above figure might be:

```
package Counter is          -- resulting object
  procedure Reset;          -- applicable function ...
  procedure Increment;
  function Current_Value return Natural;
end Counter;

package body Counter is
  Count : Natural := 0;   -- simple object
  procedure Reset is
  begin
    Count := 0;
  end Reset;
  procedure Increment is
  begin
    Count := Count + 1;
  end Increment;
  function Current_Value return Natural is
  begin
    return Count;
  end Current_Value;
end Counter;


package Max_Count is        -- resulting object
  procedure Reset;          -- applicable function ...
  procedure Increment;
  function Current_Value return Natural;
  function Max return Natural;
end Max_Count;

with Counter;
package body Max_Count is
 Max_Val : Natural := 0;   -- additional object
  procedure Reset is
  begin
    Counter.Reset;
  end Reset;
  procedure Increment is
  begin
    Counter.Increment;
    if Max_Val < Counter.Current_Value then
      Max_Val := Counter.Current_Value;
    end if;
  end Increment;
  function Current_Value return Natural is
  begin
    return Counter.Current_Value;
  end Current_Value;
  function Max return Natural is
  begin
    return Max_Val;
  end Max;
end Max_Count;
```

In this example, the objects are properly encapsulated, though, they might not have been. If, for example, the simple objects were declared in separate components from their applicable functions, the result could have been the same (although the diagram might look different). In actual practice, Ada programs are developed with a combination of encapsulated object-operation groups as well as separately declared object-operation groups. Often the lowest levels are encapsulated while the higher level and larger objects tend to be separate from their applicable function. Perhaps in the ideal case, all objects would be encapsulated with their applied function since encapsulation usually makes the process of extracting the functionality at a later time easier. This, therefore, becomes one of the guidelines revealed by this model.

If the above example were transformed to separate the functionality from each object, the following set of components might be derived:

```
generic
  type Count_Object is (<>);
package Gen_Counter is      -- resulting object
  procedure Reset;          -- applicable function ...
  procedure Increment;
  function Current_Value return Count_Object;
end Gen_Counter;

package body Gen_Counter is
  Count : Count_Object        -- simple object
        := Count_Object'First;
  procedure Reset is
  begin
    Count := Count_Object'First;
  end Reset;
  procedure Increment is
  begin
    Count := Count_Object'Succ (Count);
  end Increment;
  function Current_Value return Count_Object is
  begin
    return Count;
  end Current_Value;
end Gen_Counter;

generic
  type Count_Object is (<>);
package Gen_Max_Count is  -- resulting object
  procedure Reset;          -- applicable function ...
  procedure Increment;
  function Current_Value return Count_Object;
  function Max return Count_Object;
end Gen_Max_Count;

with Gen_Counter;
package body Gen_Max_Count is
  Max_Val : Count_Object      -- additional object
              := Count_Object'First;
  package Counter is
              new Gen_Counter (Count_Object);
```

```
    procedure Reset is
    begin
      Counter.Reset;
    end Reset;
    procedure Increment is
    begin
      Counter.Increment;
      if Max_Val < Counter.Current_Value then
        Max_Val := Counter.Current_Value;
      end if;
    end Increment;
    function Current_Value return Natural is
    begin
      return Counter.Current_Value;
    end Current_Value;
    function Max return Natural is
    begin
      return Max_Val;
    end Max;
  end Gen_Max_Count;

  with Gen_Max_Count;
  procedure Max_Count_User is
    package Max_Count is
                new Gen_Max_Count (Natural);
  begin
    Max_Count.Reset;
    Max_Count.Increment;
    ...
  end Max_Count_User;
```

Note that the end user obtains the same functionality that a user of Max_Count has, but the software now allows the primitive object Natural to be supplied externally to the algorithms that will apply to it. Further, the user could have obtained analogous functionality for any discrete type simply by pairing the general object with a different type (using a different generic instantiation).

This model is somewhat analogous to the one used in Smalltalk programming where objects are assembled from other objects plus programmer-supplied specifics. However, it is meant to apply more generally to Ada and other languages that do not have support for dynamic binding and full inheritance, features that are in general unavailable when strong static type checking is required. Instead, Ada offers the generic feature which can be used as shown here to partially offset the constraints imposed by static checking.

Applying this model to existing software means that any lines of code which represent reusable functionality must be parameterized with generic formal parameters in order to make them independent from their surrounding declaration space (if they are not already independent). Generics that are extracted by generalizing existing program units, through the removal of their dependence on external declarations, can then be offered as independently reusable components for other applications.

Unfortunately, declarative dependence is only one of the ways that a program unit can depend on its external environment. Removing the compiler-detectable declarative dependencies by producing a generic unit is no guarantee that the new unit will actually be independent. There can be dependencies on data values that are related to values in neighboring software, or even dependencies on protocols of operation that are followed at the point where a resource was originally used but which could be violated at a point of later reuse. (An example of this kind of dependency is described in the Measurement section.) To be complete, the transformation process would need to identify and remove these other types of dependence as well as the declarative dependence. Although guidelines have been identified by this study which can reduce the possibility for these other types of dependencies to enter a system, this work only concentrates on mechanisms to measure and remove declarative dependence.

More Examples

In a language with strong static type checking, such as Ada, any information exchanged between communicating program units must be of some type which is available to both units. Since Ada enforces name equivalence of types, where a type name and not just the underlying structure of a type introduces a new and distinct type, the declaration of the type used to pass information between units must be visible to both of those units. The user of a resource, therefore, is constrained to be in the scope of all type declarations used in the interface of that resource. In a language with a fixed set of types this is not a problem since all possible types will be globally available to both the resource and its users. However, in a language which allows user-declared types and enforces strong static type checking of those types, any inter-component communication with such types must be performed in the scope of those programmer-defined declarations. This means that the coupling between two communicating components increases from data coupling to external coupling (or from level two to level five on the traditional seven-point scale of Myers, where level one is the lowest level of coupling) [5].

Consider, for example, project-specific type declarations which often appear at low, commonly visible levels in a system. Resources which build upon those declarations can then be used in turn by higher level application-specific components. If a programmer attempts to reuse those intermediate-level resources in a new context, it is necessary to also reuse the low-level declarations on which they are built. This may not be acceptable, since combining several resources from different original contexts means that the set of low-level type declarations needed can be extensive and not generally compatible. This situation can occur whether or not data is encapsulated with its applicable function, but for clarity, and to contrast with the previous examples, it is shown here with the data and its operations declared separately.

For example, imagine that two existing programs each contain one of the following pairs of compilation units:

```
-- First program contains first pair:
package Vs_1 is
  type Variable_String is
    record
      Data    : String (1..80);
      Length  : Natural;
    end record;
  function Variable_String_From_User
                       return Variable_String;
end Vs_1;
```

```
with Vs_1;
package Pm_1 is
  type Phone_Message is
    record
      From : Vs_1.Variable_String;
      To   : Vs_1.Variable_String;
      Data : Vs_1.Variable_String;
    end record;
  function Phone_Message_From_User
                          return Phone_Message;
end Pm_1;

-- Second program contains second pair:
package Vs_2 is
  type Variable_String is
    record
      Data   : String (1..250) := (others=>' ');
      Length: Natural := 0;
    end record;
  function Variable_String_From_User
                          return Variable_String;
end Vs_2;

with Vs_2;
package Mm_2 is
  type Mail_Message is
    record
      From    : Vs_2.Variable_String;
      To      : Vs_2.Variable_String;
      Subject : Vs_2.Variable_String;
      Text    : Vs_2.Variable_String;
    end record;
  function Mail_Message_From_User
                          return Mail_Message;
end Mm_2;
```

Now, consider the programmer who is trying to reuse the above declarations in the same program. A reasonable way to combine the use of Mail_Messages with the use of Phone_Messages might seem to be as follows:

```
with Vs_1;
with Pm_1;
with Mm_2;
procedure User is
  Name : Vs_1.Variable_String;
  Pm : Pm_1.Phone_Message :=
              Pm_1.Phone_Message_From_User;
  Mm : Mm_2.Mail_Message :=
              Mm_2.Mail_Message_From_User;
begin
  Name := Pm.To;
  Mm.From := Name;      -- illegal
end User;
```

This will fail to compile, however, since the types Vs_1.Variable_String and Vs_2.Variable_String are distinct and therefore values of one are not assignable to objects of the other (the value of Name is of type Vs_1.Variable_String and the record component Mm.From is of type Vs_2.Variable_String).

In the above example, note that the variable string types were left visible rather than made private to make it seem even more plausible for a programmer to expect that, at least logically, the assignment attempted is reasonable. However, the incompatibility between the underlying type declarations used by Mail_Message and Phone_Message becomes a problem. One solution might be to use type conversion. However, employing type conversion between elements of the low level variable string types destroys the abstraction for the higher-level units. For instance, the user procedure above could be written as shown below, but exposing the detail of the implementation of the variable strings represents a poor, and possibly dangerous, programming style.

```
with Vs_1;
with Pm_1;
with Mm_2;
procedure Type_Conversion_User is
  Name : Vs_1.Variable_String;
  Pm : Pm_1.Phone_Message :=
            Pm_1.Phone_Message_From_User;
  Mm : Mm_2.Mail_Message :=
            Mm_2.Mail_Message_From_User;
begin
  Name := Pm.To;
  Mm.From.Data (1..80) := Name.Data;
  Mm.From.Length := Name.Length;
end Type_Conversion_User;
```

Notice that we had to be careful to avoid a constraint error at the point of the data assignment. This is one example of how attempts to combine the use of resources which rely on different context declarations is difficult in Ada.

Static type checking, therefore, is a mixed blessing. It prevents many errors from entering a software system which might not otherwise be detected until run time. However, it limits the possible reuse of a module if a specific declaration environment must also be reused. Not only must the reused module be in the scope of those declarations, but so must its users. Further, those users are forced to communicate with that module using the shared external types rather than their own, making the resource master over its users instead of the other way around. The set of types which facilitates communication among the components of a program, therefore, ultimately prevents most, if not all, of the developed algorithms from being easily used in any other program.

This study refers to declarations such as those of the above variable string types as *contexts*, and to components which build upon those declarations and which are in turn used by other components, such as the above Mail_Message and Phone_Message packages, as *resources*. Components which depend on resources are referred to as *users*. The above illustrates the general case of a context-resource-user relationship. It is possible for a component to be both a resource at one level and also a context for a still higher-level resource. The dependencies among these three basic categories of components can be illustrated with a directed graph. Figure 2 shows a graph of the kind of dependency illustrated in the example above.

A resource does not always need full type information about the data it must access in order to accomplish its task. In the above examples, it would be possible for the Mail and Phone message resources to implement their functions via the functions exported from the variable string packages without any further information about the structures of those lower level variable string types. Sometimes, even less knowledge

of the structure or functionality of the types being manipulated by a resource is required by that resource for it to accomplish its function.

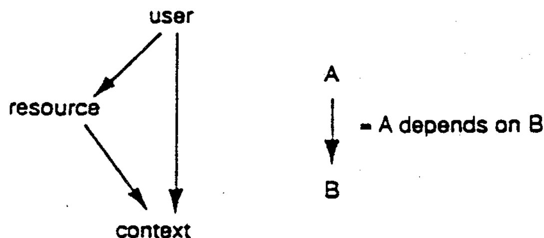

Figure 2.

A common example of a situation where a resource needs no structural or operational information about the objects it manipulates is a simple data base which stores and retrieves data but which does not take advantage of the information contained by that data. It is possible to write or transform such a resource so that the context it requires (i.e., the type of the object to be stored and retrieved) is supplied by the users of that resource. Then, only the essential work of the module needs to remain. This "essence only" principle is the key to the transformations sought. Only the purpose of a module remains, with any details needed to produce the executing code, such as actual type declarations or specific operations on those types, being provided later by the users of the resource. In languages such as Smalltalk which allow dynamic binding, this information is bound at run time. In Ada, where the compiler is obligated to perform all type checking, generics are bound at compilation time, eliminating a major source of run time errors caused by attempting to perform inappropriate operations on an object. Even though they are statically checked, however, Ada generics can often allow a resource to be written so as to free it from depending upon external type definitions.

Using the following arbitrary type declaration and a simplified data store package, one possible transformation is illustrated. First the example is shown before any transformation is applied:

```
-- context:
package Decls is
   type Typ is ...   -- anything but limited private
end Decls;

-- resource:
with Decls;
package Store is
  procedure Put (Obj : in Decls.Typ);
  procedure Get_Last (Obj : out Decls.Typ);
end Store;
```

```
package body Store is
  Local : Decls.Typ;
  procedure Put (Obj : in Decls.Typ) is
  begin
    Local := Obj;
  end Put;
  procedure Get_Last (Obj : out Decls.Typ) is
  begin
    Obj := Local;
  end Get_Last;
end Store;
```

The above resource can be transformed into the following one which has no dependencies on external declarations:

```
-- generalized resource:
generic
  type Typ is private;
package General_Store is
  procedure Put (Obj : in Typ);
  procedure Get_Last (Obj : out Typ);
end General_Store;

package body General_Store is
  Local : Typ;
  procedure Put (Obj : in Typ) is
  begin
    Local := Obj;
  end Put;
  procedure Get_Last (Obj : out Typ) is
  begin
    Obj := Local;
  end Get_Last;
end General_Store;
```

Note that, by naming the generic formal parameter appropriately, none of the identifiers in the code needed to change, and the expanded names were merely shortened to their simple names. This minimizes the handling required to perform the transformation (although automating the process would make this an unimportant issue). This transformation required the removal of the context clause, the addition of two lines (the generic part) and the shortening of the expanded names. The modification required to convert the package to a theoretically independent one constitutes a reusability measure. A user of the resource in the original form would need to add the following declaration in order to obtain an appropriate instance of the resource:

```
package Store is new General_Store (Decls.Typ);
```

Formal rules for counting program changes have already been proposed and validated [4], and adaptations of these counting rules (such as using a lower handling value for shortening expanded names and a higher one for adding generic formals) are being considered as part of this work.

The earlier example with the variable string types can also be generalized to remove the dependencies between the mail and phone message packages (resources) and the variable string packages (contexts). For example, ignoring the implementations (bodies) of the resources, the following would functionally be equivalent to those examples:

```
-- Contexts, as before:
package Vs_1 is
  type Variable_String is
    record
      Data : String (1..80);
      Len  : Natural;
    end record;
  function Variable_String_From_User
                          return Variable_String;
end Vs_1;

package Vs_2 is
  type Variable_String is
    record
      Data : String (1..250) := (others=>' ');
      Len  : Natural := 0;
    end record;
  function Variable_String_From_User
                          return Variable_String;
end Vs_2;
```

```
-- Resources, which no longer depend upon
-- the above context declarations:
generic
  type Component is private;
package Gen_Pm_1 is
  type Phone_Message is
    record
      From : Component;
      To   : Component;
      Data : Component;
    end record;
  function Phone_Message_From_User
                          return Phone_Message;
end Gen_Pm_1;

generic
  type Component is private;
package Gen_Mm_2 is
  type Mail_Message is
    record
      From : Component;
      To   : Component;
      Subj : Component;
      Text : Component;
    end record;
  function Mail_Message_From_User
                          return Mail_Message;
end Gen_Mm_2;
```

Now, the programmer who is trying to reuse the above declarations by combining the use of Mail_Messages with the use of Phone_Messages has another option. Instead of trying to combine both contexts, just one can be chosen (in this case, Vs_2):

```
with Vs_2;
with Gen_Pm_1;
with Gen_Mm_2;
procedure User is
  package Pm_1 is new
            Gen_Pm_1 (Vs_2.Variable_String);
  package Mm_2 is new
            Gen_Mm_2 (Vs_2.Variable_String);
  Name : Vs_2.Variable_String;
  Pm : Pm_1.Phone_Message :=
              Pm_1.Phone_Message_From_User;
  Mm : Mm_2.Mail_Message :=
              Mm_2.Mail_Message_From_User;
begin
  Name := Mm.From;
  Pm.To := Name;     -- now OK
end User;
```

An additional complexity is required for this example. The resources must be able to obtain component type values from which to construct mail and phone messages. Although this is not obvious from the specifications only, it can be assumed that such functionality must be available in the body. This can be done by adding a generic formal function parameter to the generic parts, requiring the user to supply an additional parameter to the instantiations as well:

```
generic
  type Component is private;
  with function Component_From_User
        return Component;
                        -- parameterless for simplicity
package Gen_Pm_1 is
  type Phone_Message is
    record
      From : Component;
      To   : Component;
      Data : Component;
    end record;
  function Phone_Message_From_User
                          return Phone_Message;
end Gen_Pm_1;
```

Although the above examples show the context, the resource, and the user as library level units, declaration dependence can occur, and transformations can be applied, in situations where the three components are nested. For example, the resource and user can be co-resident in a declarative area, or the user can contain the resource or vice versa.

This reiterates the earlier claim that, at least for the purpose of this model, it does not matter if the data is encapsulated with its applicable function, it just makes it easier to find if it is. In the programs studied, the lowest level data types, which were often properly encapsulated with their immediately available operations, were used to construct higher level resources specific to the problem being solved. It was unusual for those resources to be written with the same level of encapsulation and independence as the lower level types, and this resulted in the kind of context-resource-user dependencies illustrated above.

For example, in the case of the generalized simple data base, the functionality of the data appears in the resource while the declaration of it appears in the context. The only place where the higher-level object comes into existence is inside the user component, at the point where the instantiation is declared. If desired, an additional transformation can be applied to rectify this problem of the apparent separation of the object from its operations. Instead of leaving the instantiation of the new generic resource up to the client

software, an intermediate package can be created which combines the visibility of the context declarations with instantiations of the generic resource. This package, then, becomes the direct resource for the client software, introducing a layer of abstraction that was not present in the original (non-general) structure.

For example, the following transformation to the second example above combines the resource General_Store with the context of choice, type Typ from package Decls. The declaration of the package Object performs this service.

```
generic
  type Typ is private;
package General_Store is
  procedure Put (Obj : in Typ);
  procedure Get_Last (Obj : out Typ);
end General_Store;

package Decls is
  type Typ is ...
end Decls;

with Decls;
with General_Store;
package Object is
  subtype Typ is Decls.Typ;
  package Store is new General_Store (Typ);
  procedure Put (Obj : in Typ)
          renames Store.Put;
  procedure Get_Last (Obj : out Typ)
          renames Store.Get_Last;
end Object;

with Object;
procedure Client is
  Item : Object.Typ;
begin
  Object.Put (Item);
  Object.Get_Last (Item);
end Client;
```

Note that no body for package Object is required using the style shown. If it were preferable to leave the implementation of Object flexible, so that users would not need to be recompiled if the context used by the instantiation were to change, the context clauses and the instantiation could be made to appear only in the body of Object. An alternate, admittedly more complex, example is shown here which accomplishes this flexibility:

```
package Object is
  type Typ is  private;
  function Initial return Typ;
  procedure Put (Obj : in Typ);
  procedure Get_Last (Obj : in Typ);
private
  type Designated;
  type Typ is access Designated;
end Object;

with Decls;
with General_Store;
package body Object is
  type Typ is new Decls.Typ;
  function Initial return Typ is
  begin
    return new Designated;
  end Initial;
  package Store is new General_Store (Typ);
  procedure Put (Obj : in Typ) is
  begin
     Store.Put (Obj.all);
  end Put;
  procedure Get_Last (Obj : in Typ) is
  begin
     Store.Get_Last (Obj.all);
  end Get_Last;
end Object;
```

In the alternate example, note that the parameter mode for the Get_Last procedure needed to be changed to allow the reading of the designated object of the actual access parameter. Also, a simple initialization function was supplied to provide the client with a way of passing a non-null access object to the Put and Get_Last procedures. Normally, there would already be initialization and constructor operations, so this additional operation would not be needed. The advantage of this alternative is that the implementation of the type and operations can change without disturbing the client software. However, the first alternative could be changed in a compilation-compatible way, such that any client software would need recompilation but no modification.

It is also possible to provide just an instantiation as a library unit by itself, but this requires the user to acquire independently the visibility to the same context as that instantiation. This solution results in the reconstruction of the original situation, where the instantiation becomes the resource dependent on a context, and the user depends on both. The important difference, however, is that now the resource (the instantiation) is not viewed as a reusable component. It becomes application-specific and can be routinely (potentially automatically) generated from both the generalized reusable resource and the context of choice, while the generic from which the instantiation is produced remains the independent, reusable component. The advantage of this structure lies in the abstraction provided for the user component which is insulated from the complexities of the instantiation of the reusable generic. Since the result is similar to the initial architecture, the overall software architecture can be preserved while utilizing generic resources. The following illustrates this.

```
package Decls is
  type Typ is ...
end Decls;

generic
  type Typ is private;
package General_Store is
  procedure Put (Obj : in Typ);
  procedure Get_Last (Obj : out Typ);
end General_Store;

with Decls;
with General_Store;
package Object is new General_Store(Decls.Typ);
```

```
with Decls;
with Object;
procedure Client is
  Item : Decls.Typ;
begin
  Object.Put (Item);
  Object.Get_Last (Item);
end Client;
```

By modifying the generic resource to "pass through" the generic formal types, the user's reliance on the context can be removed:

```
generic
  type Gen_Typ is private;
package General_Store is
  subtype Typ is Gen_Typ;   -- pass the type through
  procedure Put (Obj : in Typ);
  procedure Get_Last (Obj : out Typ);
end General_Store;

package Decls is
  type Typ is ...
end Decls;

with Decls;
with General_Store;
package Object is new General_Store(Decls.Typ);

with Object;
procedure Client is
  Item : Object.Typ;
begin
  Object.Put (Item);
  Object.Get_Last (Item);
end Client;
```

Measurement

In the above examples, the context components were never modified. Resource components were modified to eliminate their dependence on context components. User components were modified in order to maintain their functionality given the now general resource components, typically by defining generic actual parameter objects and adding an instantiation. In the case of the encapsulated instantiations, an intermediate component was introduced to free the user component of the complexity of the instantiation. It is the ease or difficulty of modifying the resource components that is of primary interest here, and the measurement of this modification effort constitutes a measurement of the reusability of the components. The usability of the generalized resources is also of interest, since some may be difficult to instantiate.

Considering the above examples again, the simple data base resource Store required the removal of the context clause and the creation of a generic part (these being typical modifications for almost all transformations of this kind). In addition, the formal parameter types for the two subprograms were changed to the generic formal private type, causing a change to both the subprogram specification and body. No further changes were required.

```
-- original:
with Decls;
package Store is
  procedure Put (Obj : in Decls.Typ);
  procedure Get_Last (Obj : out Decls.Typ);
end Store;

package body Store is
  Local : Decls.Typ;
  procedure Put (Obj : in Decls.Typ) is
  begin
    Local := Obj;
  end Put;
  procedure Get_Last (Obj : out Decls.Typ) is
  begin
    Obj := Local;
  end Get_Last;
end Store;

-- transformed:
generic
  type Typ is private;                    -- change
package General_Store is
  procedure Put (Obj : in Typ);           -- change
  procedure Get_Last (Obj: out Typ);      -- change
end General_Store;

package body General_Store is
  Local : Typ;
  procedure Put (Obj : in Typ) is         -- change
  begin
    Local := Obj;
  end Put;
  procedure Get_Last (Obj: out Typ) is    -- change
  begin
    Obj := Local;
  end Get_Last;
end General_Store;
```

The Phone_Message and Mail_Message resources required the deletion of the context clause, the addition of a generic part consisting of a formal private type parameter and a formal subprogram parameter, and the replacement of three occurrences (or four, in the case of Mail_Message) of the type mark Vs_1.Variable_String with the generic formal type Component.

```
-- original:
with Vs_1;
package Pm_1 is
  type Phone_Message is
    record
      From : Vs_1.Variable_String;
      To   : Vs_1.Variable_String;
      Data : Vs_1.Variable_String;
    end record;
  function Phone_Message_From_User
                        return Phone_Message;
end Pm_1;

-- transformed:
generic
  type Component is private;      -- change
  with function Component_From_User
          return Component;       -- change
```


6109

```
package Gen_Pm_1 is
  type Phone_Message is
    record
      From : Component;            -- change
      To   : Component;            -- change
      Data : Component;            -- change
    end record;
  function Phone_Message_From_User
                          return Phone_Message;
end Gen_Pm_1;
```

Generalizing the bodies of Gen_Pm_1 and Gen_Mm_2 would involve replacing any calls to the Variable_String_From_User functions with calls to the generic formal Component_From_User function. In the case of the simple bodies shown before, this would require three and four simple substitutions, for Gen_Pm_1 and Gen_Mm_2, respectively.

In addition to measuring the reusability of a unit by the amount of transformation required to maximize its independence, reusability can also be gauged by the amount of residual dependency on other units which cannot be eliminated, or which is unreasonably difficult to eliminate, by any of the proposed transformations. For any given unit, therefore, two values can be obtained. The first reveals the number of program changes which would be required to perform any applicable transformations. The second indicates the amount of dependence which would remain in the unit even after it was transformed. The original units in the examples above would score high on the first scale since the handling required for its conversion was negligible, implying that its reusability was already good (i.e., it was already independent or was easy to make independent of external declarations). After the transformation, there remain no latent dependencies, so the transformed generic would receive a perfect reusability score.

Note that the object of any reusability measurement, and therefore, of any transformations, need not be a single Ada unit. If a set of library units were intended to be reused together then the metrics as well as the transformations could be applied to the entire set. Whereas there might be substantial interdependence among the units within the set, it still might be possible to eliminate all dependencies on external declarations.

In the above examples, one reason that the transformation was trivial was that the only operation performed on objects of the external type was assignment (except for the mail and phone message examples). Therefore, it was possible to replace direct visibility to the external type definition with a generic formal private type. A second example illustrates a slightly more difficult transformation which includes more assumptions about the externally declared type. In the following example, indexing and component assignment are used by the resource.

Before transformation:

```
-- context
package Arr is
  type Item_Array is
         array (Integer range <>) of Natural;
end Arr;

-- resource
with Arr;
procedure Clear (Item : out Arr.Item_Array) is
begin
  for I in Item'Range loop
    Item (I) := 0;
  end loop;
end Clear;

-- user
with Arr, Clear;
procedure Client is
  X : Arr.Item_Array (1..10);
begin
  Clear (X);
end Client;
```

After transformation:

```
-- context (same)
package Arr is
  type Item_Array is
         array (Integer range <>) of Natural;
end Arr;

-- generalized resource
generic
   type Component is range <>;
   type Index is range <>;
   type Gen_Array is
          array (Index range <>) of Component;
procedure Gen_Clear (Item : out Gen_Array);
procedure Gen_Clear (Item : out Gen_Array) is
begin
  for I in Item'Range loop
    Item (I) := 0;
  end loop;
end Gen_Clear;

-- user
with Arr, Gen_Clear;
procedure Client is
  X : Arr.Item_Array (1..10);
  procedure Clear is new Gen_Clear
                          (Natural,
                           Integer,
                           Arr.Item_Array);
begin
  Clear (X);
end Client;
```

The above transformation removes compilation dependencies, and allows the generic procedure to describe its essential function without the visibility of external declarations. As before, an intermediate object could be created to free the user procedure from the chore of instantiating a Clear procedure, which requires visibility to both the context and the resource. However, it also illustrates an important additional kind of dependence which can exist between a resource and its users, namely information dependence.

In the previous example, the literal value 0 is a clue to the presence of information that is not general. Therefore, the following would be an improvement over the transformation shown above:

```
generic
   type Component is range <>;
   type Index is range <>;
   type Gen_Array is
          array (Index range <>) of Component;
   Init_Val : Component := Component'First;
procedure Gen_Clear (Item : out Gen_Array);
procedure Gen_Clear (Item : out Gen_Array) is
begin
  for I in Item'Range loop
    Item (I) := Init_Val;
  end loop;
end Gen_Clear;
```

Note that the last transformation allows the user to supply an initial value, but also provides the lowest value of the component type as a default. An additional refinement would be to make the component type private which would mean that Init_Val could not have a default value. Information dependencies such as the one illustrated here are harder to detect than compilation dependencies. The appearance of literal values in a resource is often an indication of an information dependence.

A third form of dependence, called protocol dependence, has also been identified. This occurs when the user of a resource must obey certain rules to ensure that the resource behaves properly. For example, a stack which is used to buffer information between other users could be implemented in a not-so-abstract fashion by exposing the stack array and top pointer directly. In this case, all users of the stack must follow the same protocol of decrementing the pointer before popping and incrementing after pushing, and not the other way around. Beyond the recognition of it, no additional treatment of this form of dependence between components will appear in this study.

Formalizing the Transformations

The following is a formalization of the objectives of transformations which are needed to remove declaration dependence.

1. Let P represent a program unit.

2. Let D represent the set of n object declarations, $d_1 .. d_n$, directly referenced by P such that d_i is of a type declared externally to P.

3. Let $O_1 .. O_n$ be sets of operations where O_i is the set of operations applied to d_i inside P.

4. P is completely transformable if each operation in each of the sets, $O_1 .. O_n$ can be replaced with a predefined or generic formal operation.

The earlier example transformation is reviewed in the context of these definitions:

1. Let P represent a program unit.
   P = **procedure** Clear (Item : **out** Arr.Item_Array) **is** ...

2. Let D represent the set of n object declarations, $d_1 .. d_n$, directly referenced by P such that d_i is of a type declared externally to P.
   D = { Arr.Item_Array }.

3. Let $O_1 .. O_n$ be sets of operations where O_i is the set of operations applied to d_i inside P.
   O_1 =
{ indexing by integers, integer assignment to components }

4. P is completely transformable if each operation in each of the sets, $O_1 .. O_n$ can be replaced with a predefined or generic formal operation.

Indexing can be obtained through a generic formal array type. Although no constraining operation was used, the formal type could be either constrained or unconstrained since the only declared object is a formal subprogram parameter. Since component assignment is required, the component type must not be limited. Therefore, the following generic formal parts are possible:

```
type Component is range <>;
type Index is range <>;
```

followed by either:

```
type Gen_Array is array (Index) of Component;
```

or:

```
type Gen_Array is
        array (Index range <>) of Component;
```

Notice that some operations can be replaced with generic formal operations more easily than others. For example, direct access of array structures can generally be replaced by making the array type a generic formal type. However, direct access into record structures (using "dot" notation) complicates transformations since this operation must be replaced with a user-defined access function.

Application to External Software

Medium-Sized Projects

To test the feasibility of the transformations proposed, a 6,000-line Ada program written by seven professional programmers was examined for reuse transformation possibilities. The program consisted of six library units, ranging in size from 20 to 2,400 lines. Of the 30 theoretically possible dependencies that could exist among these units, ten were required. Four transformations of the sort described above were made to three of the units. These required an additional 44 lines of code (less than a 1% increase) and reduced the number of dependencies from ten to five, which is the minimum possible with six units. Using one possible program change definition, each transformation required between two and six changes.

A fifth modification was made to detach a nested unit from its parent. This required the addition of 15 lines and resulted in a total of seven units with the minimum six dependencies. Next, two other functions were made independent of the other units. Unlike the previous transformations which were targeted for later reuse, however, these transformations resulted in a net reduction in code since the resulting components were reused at multiple points within this program. Substantial information dependency which would have impaired actual reuse was identified but remained within the units, however.

A second medium-sized project was studied which exhibited such a high degree of mutual dependence between pairs of library units that, instead of selecting smaller units for generalizations, the question of non-hierarchical dependence was studied at a system level. The general conclusion from this was that loops in the dependency structure (where, for example, package A is referenced from package body B and package B is referenced from package body A) make generalization of those components difficult. The program was instead analyzed for possible restructuring to remove as much of the bi-directional dependence as practical. This was partially successful and suggests that this sort of redesign might appropriately precede other reuse analyses.

The NASA Projects

Currently, the research project is examining several spacecraft flight simulation programs from the NASA Goddard Space Flight Center. These programs are each more than 100,000 editor lines of Ada. They have been developed by an organization that originally developed such simulators in Fortran and has been transitioning to the use of Ada over the past several years. Because all the programs are in the same application domain and were developed by the same organization there is considerable opportunity for reuse. In the past, the development organization reported the ability to reuse about 20% of earlier programs when a new program was being developed in Fortran. However, since becoming familiar with Ada, the same organization is now reporting a 70% reuse rate, or better.

After gaining an understanding of the nature of the reuse accomplished in Fortran and later in Ada, and how similar or different reuse in the two languages was, we would like to test several theories about why the Ada reuse has been so much greater. We already know that the reuse is accomplished by modifying earlier components as required, and not, in general, by using existing software verbatim. Because of this reuse mode, one theory we will be testing is that the Ada programs are more reusable simply because they are more understandable.

For the current study, the programs were studied to reveal opportunities to extract generic components which, had they been available when the programs were being developed originally, could have been reused without modification. There is an additional advantage to working with this data, however, since, as mentioned above, the several programs already exhibit significant functional similarities which can be studied for possible generalization. In other words, whereas the initial discussion of generic extraction has focussed on attempts to completely free the essential function of a component from its static declaration context, this data gives examples of similar components in two or more different program contexts and therefore allows us to study the possibility of freeing a component from only its program specific context and not from any context which remains constant across programs.

This gives rise to the notion of domain-specific generic extraction as opposed to domain-independent generic extraction. Given the problems associated with extracting a completely general component, as examined earlier, a case can be made to generalize away only some of the dependence, leaving the rest in place. The additional problem, then, becomes how to determine what dependence is permissible and what should be removed. The permissible dependence would be common across projects in a certain domain, and would therefore be domain-specific while the dependence to be removed would be the problem-specific context. When reused, then, these components would have their problem-specific context supplied as generic actual parameters.

This is currently a largely manual task, since the programs must be compared to find corresponding functionality and then examined to determine the intersection of that functionality. Interestingly, on the last project the developers themselves have also been devising generic components which are instantiated only one time within that program. This implied to us that some effort was being spent to make components which might be reusable with no, or perhaps only very little, modification in the next project. We have confirmed with the developers that this is in fact the case. By comparing the results of our generalizations with those done by the developers, we find that ours have much more complex generic parts but correspondingly much less dependence on other software. This is a reasonable result, since the developers already have some idea about the context for each reuse of a given generic; what aspects of that context are likely to change from project to project and what aspects are expected to remain constant across several programs. The program-specific context, only, appears in the generic parts of the generics written by the developers, while our generalizations have generic parts which contain declarations of types and operations which apparently do not need to change as long as the problem domain remains the same. In other words, when our generic parts are devised by analyzing only a single instance of a component, we cannot distinguish between program-specific and domain-specific generalizations.

One interesting question we would like to answer is whether we can derive the generic part that makes the most sense within this domain by comparing similar components from different programs and generalizing only on their differences, leaving the software in the intersection of the components unchanged. In this way, a component would be derived which would not be completely independent but, like the developer-written generics, would be sufficiently independent for reuse in the domain. Then, a comparison with the generics developed within the organization would be revealing. If the generics are similar then our process might be useful on other parts of the software that have not yet been generalized by the developers. However, if they differ greatly, it would be useful to characterize that difference and

understand what additional knowledge must be used in generalizing the repeated software. Unfortunately, there is not enough reuse of the developer's generics yet to make this final comparison but a project is currently in progress which should supply some of this data.

The following example illustrates the complexity of the generic parts which were required to completely isolate a typical unit from its context. Here, the procedure Check_Header was removed from a package body and generalized to be able to stand alone as a library level generic procedure.

```
generic
  type Time is private;
  type Duration is digits <>;
  with function Enable return Boolean;
  type Hd_Rec_Type is private;
  with procedure Set_Start
       (H : in out Hd_Rec_Type; To : Duration);
  with function Get_Start
       (H : Hd_Rec_Type) return Duration;
  with procedure Set_Stop
       (H : in out Hd_Rec_Type; To : Duration);
  with function Get_Stop
       (H : Hd_Rec_Type) return Duration;
  type Real is digits <>;
  with function Get_Att_Int
       (H : Hd_Rec_Type) return Real;
  with function Conv_Time
       (D_Float : Duration) return Duration;
  Header_Rec : in out Hd_Rec_Type;
  Goesim_Time_Step : in out Duration;
  with function Seconds_Since_1957
       (T : in Time) return Duration;
  with procedure Debug_Write (Output : String);
  with procedure Debug_End_Line;
  type Direct_File_Type is limited private;
  with procedure Direct_Read
       (File : Direct_File_Type);
  with procedure Direct_Get
       (File : in  Direct_File_Type;
        Item : out Hd_Rec_Type);
  with function Image_Of_Base_10
       (Item : Duration) return String;
  with procedure Header_Data_Error;
procedure Check_Header_Generic
  (Simulation_Start_Time : in Time;
   Simulation_Stop_Time  : in Time;
   Simulation_Time_Step  : in Duration;
   History_File : in out Direct_File_Type);
```

The instantiation of this generic part is correspondingly complex:

```
procedure Check_Header_Instance is new
                            Check_Header_Generic
   (Abstract_Calendar.Time,
    Abstract_Calendar.Duration,
    Debug_Enable,
    Attitude_History_Types.Header_Record,
    Set_Start,
    Get_Start,
    Set_Stop,
    Get_Stop,
    Utilities.Read,
    Get_Att_Hist_Out_Int,
    Converted_Time,
    History_Data.Header_Rec,
    History_Data.Goesim_Time_Step,
    Timer.Seconds_Since_1957,
    Error_Collector.Write,
    Error_Collector.End_Line,
    Direct_Mixed_Io.File_Type,
    Direct_Mixed_Io.Read,
    Get_From_Buffer,
    Image_Of_Base_10,
    Raise_Header_Data_Error);
```

In contrast, a typical generic part on a unit which was developed and delivered as part of the most recent completed project by the developers themselves is shown here:

```
with Css_Types;
generic
  Number_Of_Sensors : Natural :=
       Css_Types.Number_Of_Sensors;
  with function Initialize_Sensor
     return Css_Types.Css_Database_Type is <>;
package Generic_Coarse_Sun_Sensor is
  ...
```

Note that by allowing the visibility of Css_Types, the generic part was simplified. Being unfamiliar with the domain, had we attempted to generalize Coarse_Sun_Sensor by examining only the non-generic version of a corresponding component in another program we would not be able to tell whether the dependence on Css_Types was program-specific or domain-specific. Here, however, the developer leads us to believe that Css_Types is domain-specific while the number of sensors and sensor initialization is program specific.

Guidelines

The manual application of the principles and techniques of generic transformation and extraction has revealed several interesting and intuitively reasonable guidelines relative to the creation and reuse of Ada software. In general, these guidelines appear to be applicable to programs of any size. However, the last guideline in the list, concerning program structure, was the most obvious when dealing with medium to large programs.

- Avoid direct access into record components except in the same declarative region as the record type declaration.

Since there is no generic formal record type in Ada (without dynamic binding such a feature would be impractical) there is no straightforward way to replace record component access with a generic operation. Instead, user-supplied access functions are needed to access the components and the type must be passed as a private type. This is unlike array types for which there are two generic formal types (constrained and unconstrained). This supports the findings of others which assert that direct referencing of non-local record components adversely affects maintainability [6].

- Minimize non-local access to array components.

Although not as difficult in general as removing dependence

on a record type, removing dependence on an array type can be cumbersome.

• Keep direct access to data structures local to their declarations.

This is a stronger conclusion than the previous two, and reinforces the philosophy of using abstract data types in all situations where a data type is available outside its local declarative region. Encapsulated types are far easier to separate as resources than globally declared types since the operations are localized and contained.

• Avoid the use of literal values except as constant value assignments.

Information dependence is almost always associated with the use of a literal value in one unit of software that has some hidden relationship to a literal value in a different unit. If a unit is generalized and extracted for reuse but contains a literal value which indicates a dependence on some assumption about its original context, that unit can fail in unpredictable ways when reused. Conventional wisdom applies here, and it might be reasonable to relax the restriction to allow the use of 0 and 1. However, experience with a considerable amount of software which makes the erroneous assumption that the first index of any string is 1 has shown that even this can lead to problems.

• Avoid mingling resources with application specific contexts.

Although the purpose of the transformations is to separate resources from application specific software regardless of the program structure, certain styles of programming result in programs which can be transformed more easily and completely. By staying conscious of the ultimate goal of separating reusable function from application declarations, whether or not the functionality is initially programmed to be generic, programmers can simplify the eventual transformation of the code.

• Keep interfaces abstract.

Protocol dependencies arise from the exportation of implementation details that should not be present in the interface to a resource. Such an interface is vulnerable because it assumes a usage protocol which does not have to be followed by its users. The bad stack example illustrates what can happen when a resource interface requires the use of implementation details, however even resources with an appropriately abstract interface can export unwanted additional detail which can lead to protocol dependence.

• Avoid direct reference to package Standard.Float

Even when used to define other floating point types, direct reference to Float establishes an implementation dependence that does not occur with anonymous floating point declarations. Especially dangerous is a direct reference to Standard.Long_Float, Standard.Long_Integer, etc., since they may not even compile on different implementations. Some care must also be taken with Integer, Positive, and Natural, though in general they were not associated with as much dependence as Float. Note that fixed point types in Ada are constructed as needed by the compiler. Perhaps the same philosophy should have been adopted for Float and Integer. Reference to Character and Boolean is not a problem since they are the same on all implementations.

• Avoid the use of 'Address

Even though it is not necessary to be in the scope of package System to use this attribute, it sets up a dependency on System.Address that makes the software non-portable. If this attribute is needed for some low-level programming than it should be encapsulated and never be exposed in the interface to that level.

• Consider the inter-component dependence of a design

By understanding how functionally-equivalent programs can vary in their degree of inter-component dependence, designers and developers can make decisions about how much dependence will be permitted in an evolving system, and how much effort will be applied to limit that dependence. For system developments which are expected to yield reusable components directly, a decision can be made to minimize dependencies from the outset. For developments which are not able to make such an investment in reusability, a decision can be made to allow certain kinds of dependencies to occur. In particular, dependencies which are removable through subsequent transformation might be allowed while those that would be too difficult to remove later might be avoided. A particularly cumbersome type of dependence occurs when two library units reference each other, either directly or indirectly. This should be avoided if at all possible. By making structural decisions explicitly, surprises can be avoided which might otherwise result in unwanted limitations of the developed software.

Acknowledgements

This work was supported in part by the U.S. Army Institute for Research in Management Information and Computer Science under grant AIRMICS-01-4-33267, and NASA under grant NSG-5123. Some of the software analysis was performed using a Rational computer at Rational's eastern regional office in Calverton, Maryland.

References

1. Basili, V. R. and Rombach, H. D. Software Reuse: A Framework. In preparation.

2. Basili, V. R. and Rombach, H. D. The TAME Project: Towards Improvement- Oriented Software Environments. IEEE Transactions on Software Engineering, SE-14, June 1988.

3. Funk & Wagnalls, Standard College Dictionary, New York, 1977.

4. Myers, G. Composite/Structured Design, Van Nostrand Reinhold, New York, 1978.

5. Dunsmore, H.E. and Gannon, J.D. Experimental Investigation of Programming Complexity. In Proceedings ACM/NBS 16th Annual Tech. Symposium: Systems and Software, Washington D.C., June 1977.

6. Gannon, J.D., Katz, E. and Basili, V.R. Characterizing Ada Programs: Packages. In Proceedings Workshop on Software Performance, Los Alamos National Laboratory, Los Alamos, New Mexico, August 1983.

John W. Bailey is a Ph.D. candidate at the University of Maryland Computer Science Department. He is a part-time employee of Rational and has been consulting and teaching in the areas of Ada and software measurement for seven years. In addition to Ada and software reuse, his interests include music, photography, motorcycling and horse support. Bailey received his M.S. in computer science from the University of Maryland, where he also earned bachelor's and master's degrees in cello performance. He is a member of the ACM.

Victor R. Basili is a professor at the University of Maryland, College Park's Institute for Advanced Computer Studies and Computer Science Department. His research interests include measuring and evaluating software development. He is a founder and principal of the Software Engineering Laboratory, which is a joint venture among NASA, the University of Maryland and Computer Sciences Corporation. Basili received his B.S. in mathematics from Fordham College, an M.S. in mathematics from Syracuse University and a Ph.D. in computer science from the university of Texas at Austin. He is a fellow of the the IEEE Computer Society and is editor-in-chief of IEEE Transactions on Software Engineering.