View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NASA Technical Reports Server

The Software-Cycle Model for Re-Engineering and Reuse

John W. Bailey*
Victor R. Basili
The University of Maryland
Department of Computer Science
College Park, Maryland 20742

*also consultant with Rational, 6707 Democracy Blvd., Bethesda, Maryland 20817

S4-61
N93-17165
/5

Abstract

This paper reports on the progress of a study which will contribute to our ability to perform high-level, component-based programming by describing means to obtain useful components, methods for the configuration and integration of those components, and an underlying economic model of the costs and benefits associated with this approach to reuse. One goal of the study is to develop and demonstrate methods to recover reusable components from domain-specific software through a combination of tools, to perform the identification, extraction, and re-engineering of components, and domain experts, to direct the application of those tools. A second goal of the study is to enable the reuse of those components by identifying techniques for configuring and recombining the re-engineered software. This component-recovery or software-cycle model addresses not only the selection and re-engineering of components, but also their recombination into new programs. Once a model of reuse activities has been developed, the quantification of the costs and benefits of various reuse options will enable the development of an adaptable economic model of reuse, which is the principal goal of the overall study. This paper reports on the conception of the software-cycle model and on several supporting techniques of software recovery, measurement and reuse which will lead to the development of the desired economic model.

Motivation and Scope

Motivation for the development of an expert-assisted but highly structured and highly automatable model of software information capture and reuse stems in part from the recognition of the difficulty of using purely programming component-based approaches to reuse libraries. For certain kinds of objects and components a strict programming component-based library is adequate. The success of object-oriented and object-based approaches have been the most notable in this regard. However, the inability for such libraries to capture a sufficient amount knowledge to dramatically reduce subsequent software development costs in a general and problem-independent way has also been observed. On the other hand, models of software reuse which utilize domain experts in pervasive and undirected ways are also unlikely to provide a complete solution due to the large amount of responsibility and effort which is centralized in the contribution of such experts. The present work provides a structured model of information identification and reuse which is both feasible and suitable for further development and refinement.

Using the Ada language, this paper provides examples of techniques for choosing, re-engineering, and recombining components into programs. It also describes rudimentary methods for quantifying the effort to extract reusable components from existing programs as well as the effort to recombine them into new programs. It does not include the cataloging and retrieval of components, nor does it include a mechanism to quantify reusability based on empirically-derived frequency-of-use measures. It does model a proposed cycle of software development, use, re-engineering, and reuse, but it does not attempt to model other aspects of reuse within a software development environment, such as pure knowledge and experience. Other recent research papers and technical reports have covered this larger scope [Basili and Rombach], [Basili and Caldiera].

Introduction

Any component of software is seen to be composed of many functional and declarative details, some of which pertain to the specific problem being solved by the program containing that component, some of which pertain to the general application domain of the containing program, and some of which pertain to neither the problem nor the

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise or republish, requires a fee and/or specific permission.

©1991 ACM 0-89791-445-7/91/1000-0267 $1.50

PRECEDING PAGE BLANK NOT FILMED

10005788L

domain, but rather define the essence of the component's function in the abstract. Therefore, to direct the selection and re-engineering of components of software, three levels of functional specificity of the software which constitutes any component are defined: 1) problem-specific details which would be likely to differ between this and another similar application in the same domain, 2) domain-specific details which are not likely to differ between this and another similar application in the same domain but which would be unlikely to be appropriate outside of this domain, and 3) essential aspects which comprise the abstract functional core of the component and without which the component would be meaningless.

The three levels cannot be absolutely defined, nor can a given detail be deterministically assigned to a level, since from different points of view, a given detail could be thought of as belonging to different levels of specificity. Two analyses of a given component could possibly identify different sets of details at each of the three levels. However, an analysis of a candidate component for the purpose of directing the re-engineering and reuse processes must assign each identifiable detail to one of the three levels.

Once specificity levels have been assigned to all details of a candidate component, a measurement of the effort required to remove each of the problem-specific details is obtained in order to estimate the total effort to generalize the component for reuse within its domain. Further, a measurement of the effort required to remove each of the domain-specific details is obtained in order to estimate the total effort to generalize the component for reuse in other domains. If these measurements show the cost-effectiveness of either of these generalizations, then the candidate component is suitably generalized and placed in either a domain-specific or domain-independent repository, as is appropriate.

In order to assign specificity levels to all the constituent details of a candidate component, domain experts may have to be consulted. However, automation to support the identification of the details and to support the component generalization through their removal can be used to streamline the process. Further, there may be ways to capture the domain experts' decisions and the reasons for them, in order to partially automate or support any subsequent decision making which follows similar patterns.

To support the generalization process and its quantification, three styles of software component reuse which are currently being practiced are identified and examined for their adaptability to the model. These reuse styles are termed *layered*, *tailored*, and *generated* reuse. Examples illustrating them, and demonstrating how they are related by an underlying dimension of generality, are shown.

Along with these examples, proposals are given for how to measure the amount of re-engineering required to derive components suitable for the different methods of reuse, as well as the amount of effort required to recombine components using the different methods. As effort is expended to make a component more general, more opportunities to reuse it become available. However, each of those reuse opportunities will have to resupply the specifics required for the reusable component to perform its function in the new context, implying an amount of reuse effort which is proportional to the degree of generality of the component.

Therefore, an economic equation presents itself, which is how to optimize the sometimes competing factors of generalization effort, reuse effort, and breadth of utility. The solution to this equation will have to wait until more work is done on the probability of reuse for a given generalization, and other factors. Rather hard questions figure in to this equation, such as the cost-benefit of constraining a solution to take advantage of an available component (which amounts to establishing and following standards) as opposed to developing a more suitable one, and even the cost of classifying, storing and retrieving components. Developing a framework for an economic model which captures these factors is the first step to a greater understanding of these issues. The last section relates the activities defined in the software-cycle process model to this economic model of reuse.

The Software-Cycle Model

This section describes the model of software development which underlies this study. The model proposes the recycling of existing software into components which can be combined into new programs. This proposed *software cycle* takes place in the context of a software development organization and allows effort already applied to the creation of previous programs to be recaptured and used to reduce the effort needed to create new programs. This software-cycle model is consistent with models of experience capture and flow within a development organization as described by [Basili and Rombach] and [Basili and Caldiera]. It describes in detail, and proposes an implementation for, one aspect of the more comprehensive experience factory described in those studies.

The software-cycle model is so-named to describe the flow of information and experience, in the form of software, into newly developed programs where it can be recovered and packaged for efficient reuse in subsequently developed software programs. The capture and reuse of information at the delivery point of the conventional software lifecycle is clearly not the only time at which such information is

10005788L

accessible. However, this approach is chosen because at the time that software is delivered, the information is packaged in a concrete form (software programs) which can be analyzed and manipulated. Also, a substantial amount of information may be available from previously-developed programs which is not recorded in any form other than the delivered software. Further, by instituting an approach which applies effort to capture reusable information at this stage, the software development organization has the choice to separate the information recovery and repackaging from the effort to develop the software, and to conduct those activities independently and in parallel. So, for pragmatic reasons, the present model of information flow in a software development organization uses developed software as the main source for recoverable information. (See also [Caldiera and Basili].)

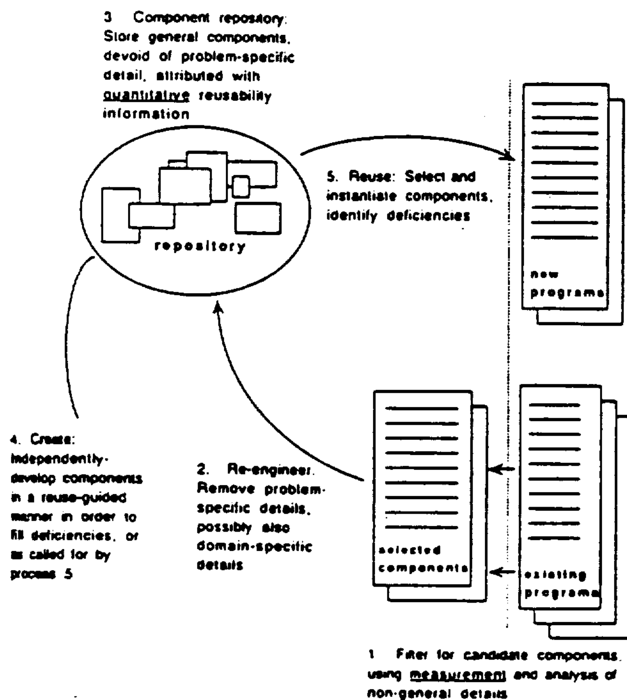

Figure 1 The processes involved in the Software-Cycle Model

As shown in Figure 1, existing programs are examined for candidate reusable components. For the purpose of this study, a component can be any definable portion of software. Obvious examples are individual, or sets of, subroutines, subprograms, functions, paragraphs, packages, or other structuring features of the software language in use. A re-engineered component can be any of these, although it can also be nothing more than a template or a set of instructions for a software generation routine.

A re-engineered component can be intended either for reuse only within a particular domain or reuse across many domains. If a component is only intended for reuse within a domain, its re-engineering seeks to remove any problem-specific details from it, but to allow any domain-specific details to remain. Such components are termed *domain-specific components*. If a component is intended for reuse across domains, however, then its re-engineering would attempt to remove all domain-specific details as well as the problem-specific details, leaving only essential function. This kind of component is termed a *domain-independent component*. Leaving a component insufficiently general to be used across domains obviously limits the number of opportunities it might enjoy for reuse. However, there are significant compensating advantages. A domain-specific component retains more details which then do not have to be resupplied by the reuse client. Also, the generalization effort to reach only problem-independence is usually less than the generalization effort required to reach domain-independence. So, by accepting a constrained reuse scope, a component can be easier to generalize as well as easier to reuse.

A candidate component for re-engineering is one which has identifiable problem-specific or domain-specific details and which can be feasibly re-engineered to eliminate the presence of some or all of those details. A domain expert may be needed to differentiate between problem-specific and domain-specific details, and measurement of the estimated generalization effort is needed to determine the feasibility of the re-engineering. Some components may be candidates to yield a domain-specific component after re-engineering but not a domain-independent component. Other components may be candidates to yield domain-independent components (possibly in addition to domain-specific components), while still others may not be good candidates to yield either category of reusable component.

The goal of reuse re-engineering is to be able to isolate and then to replace the problem-specific and/or the domain-specific aspects of a component so that it can be made to operate in different contexts. A component might be viewed as a blend of general function, which defines its essence, and specific function which relates to the current context or declarations on which the general function is performed. This is shown graphically in Figure 2a. The general function, shown in light grey, is that which is essential to the component or that which defines the nature of the component. The specific function, shown in dark grey, can either be problem-specific or domain-specific. As mentioned, it may be necessary to consult domain experts to distinguish between a problem-specific detail and a domain-specific detail. However, given a sufficient body of experience, it may be possible to predict the specificity of a detail via a predictive function that is tailored by previous expert decisions, or by statistical

10005788L

analyses of several similar components in the same domain.

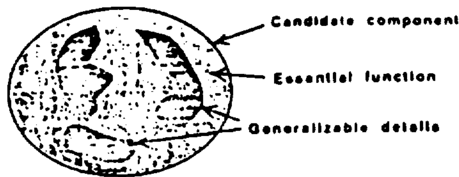

Figure 2a.

Typical candidate component, showing that it is a combination of essential function which defines the component and problem-specific or domain-specific details which can potentially be generalized in order to re-engineer the component into a more reusable one.

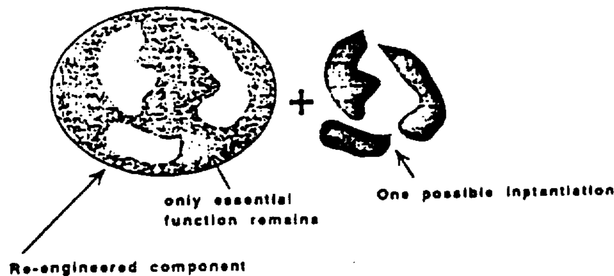

Figure 2b.

After re-engineering, the essential functionality remains in the reusable component but problem-specific or context-specific details are eliminated and become the responsibility of the reuser to provide. One possible instantiation could result in the original component again, but many other instantiations are now possible

Figure 2b shows an imaginary candidate component which contains both essential function, which is general, and specific details which, if altered, could allow the component to contribute its functionality in different contexts. These specific details, shown in dark grey, have been removed from the body of the component to signify that they are now viewed as only one of potentially many possible instantiations of the remaining, general component. The re-engineering process of the software-cycle model seeks to locate and remove these non-general aspects (either only the problem-specific aspects or, possibly, the domain-specific aspects as well) and to relegate them to the responsibility of the reuser as part of the component's instantiation. The techniques for the removal of these details are discussed as part of the section on re-engineering techniques which follows. It will be shown there that the re-engineered component does not need to be expressed in the programming language of the original candidate component which was used to produce it. It might be a pre-processable component or a component generator which can be used to produce components when necessary. In these cases, it is the template or the generator that is reusable, since any subsequently required components would be produced on demand and would not, themselves, be considered reusable.

Separated and re-engineered (generalized) components are stored in a repository to be made available to the developers of new software. Similar to the process of consulting domain experts when categorizing the details which need to be generalized out of candidate components, repository experts may have to be consulted to assist in the location and instantiation of required components in the repository. Repository experts could possibly choose from among various schemes to satisfy the needs of a developer. Certain choices might provide more utility but might come with more restrictions or limitations of options. Also, the repository expert might choose from different methods to arrive at functionally the same result to the requesting developer, for example by either generating the software or by providing a tailorable component.

Components in the repository are attributed with measurement information describing the expected effort to instantiate them for reuse. In many cases, this instantiation becomes the responsibility of the reusing developer, for example when the component is already a structural component in the developer's language of choice and simply must be supplied with actual parameters to serve the developer's need. In other cases, the instantiation can be the responsibility of the repository expert, who might have to produce components for the developer from templates, rules, instance specifications, and generator programs. In either case, the measurement attribute of a component will guide its users when deciding whether to select it or not, and how much effort to expect to expend configuring it for reuse.

A request for software components might be unfillable given the current state of a repository. In this case, the repository experts can work with the developer to design and create a new component which will not only serve the current need but which will become an instant candidate for insertion into the repository, with a minimum of re-engineering. Or, gaps in the capabilities of the repository can be identified by the experts prior to a specific need, and special developments can be guided, specifically for the purpose of supplying components to fill those gaps. In the software-cycle model, any new development is done with reuse in mind, specifically with an eye toward further populating the component repository.

Neither of these last two topics, the selection of components from a repository and the direct development of components rather than through re-engineering, are currently part of the study. They are mentioned here in order to complete the software cycle depicted in Figure 1.

10005788L

The major emphases of the study are the identification of candidate reusable components from among existing software, the re-engineering of those components to improve their generality, the measurement of those processes, and the development of an economic model which can assist an organization in optimizing its software cycle costs.

Reuse Modes and Methods

By studying the dependencies among software elements, 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 the first component would not normally be reusable in another program where the second 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 reused in any context (ignoring for the moment whether or not it performs any useful purpose in that context). The issue of software independence is at the heart of this study.

It will be seen that increased independence of a software component often comes at the cost of functionality. The ideal software reuse re-engineering process would provide a means of preserving all of the function or utility of a component while also making it independent of problem-specific or domain-specific details. However, this is not possible in most cases since some of the desired functionality is likely to be captured by those specific details, and removing the details will remove that functionality. This study describes a compromise solution, which is first to generalize a component, and then to systematize the means to configure it in order to restore the specific function required in a particular context of reuse.

A scheme to maintain generalized, reusable components in a repository, in addition to a means of configuring them in different ways for different domains or contexts, enables a repository with a manageable number of components to be described. Without the ability to instantiate a given component in different ways for different usages, a repository would have to contain many times as many assets in order to serve the same need. In order to avoid this problem, this work recommends storing fewer components, each of which is sufficiently general to be able to operate in various contexts, and then providing methods to instantiate them to provide functionality in those contexts.

By examining existing successes in software reuse, it can be seen that there are three different but related ways of making software components which are general and independent, and yet which remain capable of being instantiated with problem-specific details. An important premise of this work is that software which is general in these ways does not necessarily need to be developed directly. Instead, it is often possible to re-engineer existing software so that it achieves the necessary independence.

For this study, the three modes are termed *layered*, *tailored*, and *generated*. Each mode describes components which can be combined to develop larger programs. However, a tailored component can be made more flexible and general than a layered component and a generated component can be the most flexible and general of all. On the other hand, a layered component is the easiest to reuse, requiring the least effort on the part of the client to incorporate it into a program, while a generated component is the most difficult to reuse.

What all of these techniques strive for is the absence of dependence from the reused software on external declarations, which would hamper the generality of the software. In other words, a component of reusable software should ideally not be expected to "know" about declarations and other components which are problem-specific. A reusable resource which requires the reuser to also include other common denominator components, which contain needed declarations, is not as reusable as one which has no such requirements.

Within the confines of a single domain, however, certain dependencies can be tolerated, since the users can be expected to guarantee the minimum required declaration space across all occurrences of reuse of a component. This result opens up vast new ranges of possibilities, since the generality of a component need no longer be absolute but rather need only be general with respect to a certain domain or domains. No expectation of generality within other domains is maintained. Domain-specific reusability implies a certain amount of built-in dependence whereas wide-scale reusability or generality precludes this possibility. By allowing domain-specific constraints, the possibilities for identifying reusable components expand enormously but the breadth of applicability for each component is limited to that domain.

Layered Reuse

Layered reuse is used to describe the case where reusable functions or operations are viewed simply as abstract primitives which are callable from within the language of the client. A math library, probably the most commonly cited example of reuse, and one which is often viewed as an ideal, is an example of layered reuse. Analogous to a math package, other common examples are packages of utilities which operate on universal types or concepts, such as string handling utilities and time utilities. Other successes in layered software reuse include user interface

or I/O toolkits, graphical display toolkits, runtime kernels, and layered network protocol software.

Layered reusability is often viewed as the goal for a library of reusable components, where a sufficiently rich set of abstract operations would be available to an applications programmer in order to minimize the effort required to generate a new system. In addition to the previously mentioned independence from other components, an additional recommendation for the success of a layered component is that the data on its interface be expressed in terms of standard types. This restriction allows the client software to communicate with the reusable component without the additional complexity of adhering to specific non-standard types. One reason that a math library is so inherently reusable, for instance, is that real numbers are a universal way of expressing the values used by and returned by the mathematical functions in a library. Any language which supports real numbers can make available a corresponding set of mathematical functions.

However, unlike the portability enjoyed when restricting one's domain to a universal concept such as real numbers, a considerable amount of software which might otherwise be available for reuse is written to operate on problem-specific types and data structures. This is the case whether those types are named and declared as in Pascal or Smalltalk, are common data areas as in Fortran, or are merely locations in memory as in assembly language. Components can still be written in a layered manner but in these cases they typically depend so heavily on specific data structures that they are limited to being reused only where identical data structures or other operands are present. It is not always possible to parameterize a component with respect to all of its assumptions about context. Because of these limitations on the applicability of a layered component, constructing comprehensive reusable libraries of them in languages such as Ada has been harder than might have been expected.

Tailored Reuse

Another category of successful reuse is tailored reuse, where configuration of the reusable software is required in order to allow it to interoperate properly with the client software. A familiar example of such reuse is seen with database management systems which require tailoring in order to handle records of the user-defined structures. Simpler examples of tailored reuse are generic data structures which allow the client software to create stacks, queues, lists, etc., of application-specific types or to search through or sort objects of those types. Still other examples of tailored reuse are forms management systems which are customized by parameterization, expert systems which must be initialized with rules, spreadsheets which must be supplied with formulas, and statistics packages which must be provided with data sets and programs to achieve the desired results.

Tailoring in this way is accomplished before the component is called, but it happens automatically at execution time as part of the language behavior. Whereas in layered reuse a client simply calls a component with the proper parameters, tailored reuse implies a two-step process where a component is first molded to the specific configuration required by the current context and is then called to perform its function.

The generic feature of Ada allows certain kinds of tailoring, in the form of generic parameterization, to be accomplished. Because of the static checking enforced by Ada, however, only a limited amount of parameterizations are possible. Other languages have different mechanisms for accomplishing this parameterization. Most notably, assembly languages employ very flexible macro expansions which can be quite powerful. However, object-oriented languages have traditionally used a more flexible form of layering (full inheritance) while overlooking the possibility for component parameterization. (Future revisions to C++, however, are expected to include a template mechanism to allow within-language tailoring [Ellis and Stroustrup].)

Generated Reuse

The third category of reuse, generated reuse, occurs when the reusable software is used as a generator program rather than being incorporated directly into the final application. The required software is emitted as a result of the generator program operating on input tables or files. Typically, only the generator and not the generated software is reused. The generated software is regenerated, as opposed to being modified directly, if changes are required. Whereas layered and tailored reuse take advantage of language-supported features (subprograms and generics in the case of the Ada language) generated reuse requires additional tooling to accomplish a kind of tailoring which is external to the implementation language.

A common example of generated reuse, which perhaps stretches the definition somewhat, is a compiler, which accepts files of a high-order language and emits software in a machine-executable form. One reason that it may seem unconventional to think of a compiler as reusable software is that its output is not directly manipulated or even observed by the compiler's users. Nevertheless, it fits the definition here for generated reuse (which could be thought of as a batch form of tailored reuse).

Other common examples, where the generated output is more likely to be manipulated or at least observed by the

10005788L

users of the generator, are fourth-generation languages, user interface generators, test case generators, parser generators and table-driven forms management systems. At least one large Ada development is making substantial use of generated reuse in an MIS system development, through the use of a specially-developed generator [AIC].

Table 1 is a summary of the modes of software reuse described and the examples mentioned for each.

Layered:
Math libraries
Common utilities packages
User interface or I/O toolkits
Graphics kernel systems
Runtime kernels
Network layered software

Tailored:
Database management systems
Forms management systems (runtime configured)
Expert systems
Spreadsheets
Statistics packages
Generic data structures

Generated:
Forms management systems (file driven)
User interface generators
Test-case generators
High-order languages
Fourth-generation languages
Parser generators
MIS systems

Table 1. Reuse Modes and Examples

The distinctions between these categories can sometimes become blurred. For example, whether a reusable package is configured at run time by parameterization (tailored) or in advance by tables such that it emits a separate program (generated) may not be of any real consequence. In fact, the examples given in one category often have analogs which exist in the other category. For example, forms management systems already exist in both generated and tailored versions. Although parser generators are typically generated components, since they are stand-alone grammar-driven programs which emit desired software, they could instead be incorporated into the end-product and re-emit their parsers on the fly. The obvious reason not to do this is for efficiency of repeated use of the same output. However, an interpreter for a language can be thought of as a compiler which is configured to perform as tailorable software. In this case, the run-time efficiency is traded off for the flexibility of being able to alter the "parameterization" (the interpreted program) quickly and easily.

A Simple Example

As a simple example of how a low-level component can be viewed as a generalizable layer of function, consider the following error-reporting routine.

```
with Text_Io;
procedure Gyro_Speed_Error is
begin
   Text_Io.Put_Line ("Error: The gyros are not up to speed.");
end Gyro_Speed_Error;
```

This highly specific routine represents one end of the generality scale. It is easy to use, requiring a simple parameterless call, but might not be likely to be widely called upon within a program. There are three observable details within this unit: 1) the use of Text_Io.Put_Line to report the error message, 2) the use of the standard output device to display the error, and 3) the choice of the literal string to be displayed.

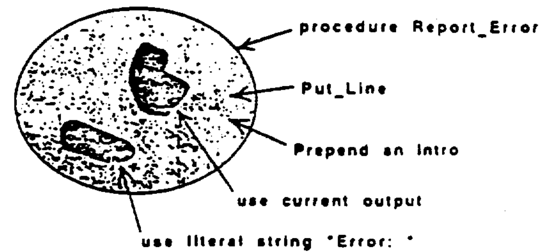

Figure 3a.

In the example from the text, procedure Report_Error was seen to be composed of four decisions. Two are considered part of the essential functionality and two are considered to be problem-specific details

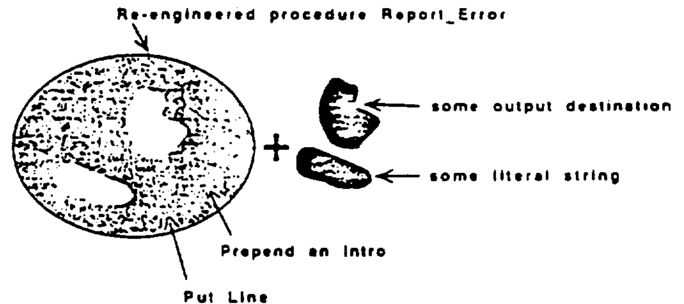

Figure 3b.

The re-engineered version of Report_Error shows the two problem-specific details removed from the component, to be supplied by the re-user. The intrinsic functional aspects of the component remain. Other interpretations of the re-engineering decisions to be applied could possibly remove one of these, as well.

10005788L

A consultation with a domain expert might result in our choice to parameterize the exact error message to be reported, which might yield the more sensible reporting routine, shown below.

```
with Text_Io;
procedure Report_Error (Message : String) is
begin
  Text_Io.Put_Line ("Error: " & Message);
end Report_Error;
```

This version of the unit is depicted in Figure 3a. Had we performed the transformation without expert consultation we might have simply parameterized the entire message. However, in our hypothetical problem domain we will assume that the expert recommended retaining a hard-coded standard prefix in order to facilitate the post-processing of the log file. Also, this generalization has cost us the part of the original functionality which spelled out the exact error message. Since the client must now supply this string, we have increased the effort to use the unit by making it more general.

The generalization of a value (a string value in this case) is the easiest kind of transformation since it can be performed with a simple value parameter. Since the parameter type is language-defined (type String) there is no further complexity to exposing this parameter in the procedure interface. Also, the effort to configure the component amounts to simply defining the error message string as a parameter. Again, this kind of reuse is the easiest.

The procedure above still assumes that the user intends the message to be written to the current output device using Put_Line. That constitutes part of the retained functionality of this component. In the process, we have also added the detail that the standard prefix "Error: " will always appear.

Additional consultation with a domain expert might reveal that the assumed use of the standard output device is another problem-specific detail. A later reuser of this component who was working on a different problem in the same domain might not want to be bound by that assumption. Again, Ada provides a simple way to parameterize the component so that users can specify the output device. Again, however, this generalization comes at the cost of functionality. In this case, the functionality which is lost is the assumption is that the current output device is to be used. Default parameters can sometimes provide an opportunity to restore such assumptions while retaining the generality, as will be shown later. The parameterized version of the unit which follows removes the assumption of using the current output device but retains the function of writing the literal string "Error: " followed by the caller's message.

```
with Text_Io;
procedure Report_Error
        (Message : String;
         On_Device : Text_Io.File_Type) is
begin
  Text_Io.Put_Line (On_Device, "Error: " & Message);
end Report_Error;
```

Notice that the user is now required to do additional work. Instead of simply providing the error message, the desired output device or file must be provided. That decision has shifted from the component to the (re)user. Again, this is a form of value parameterization, the easiest form of both generalization and reuse configuration.

An additional part of the functionality of the component is the literal string prepended to the caller's message. As shown below, this could also be parameterized, again removing that specific functionality but generalizing the component on that behavior. This requires yet one more piece of information from the user as part of the information needed for this component to perform its work, however once again it is a low-cost value parameterization.

```
with Text_Io;
procedure Report_Error
        (Message : String;
         Intro : String;
         On_Device : Text_Io.File_Type) is
begin
  Text_Io.Put_Line (On_Device, Intro & Message);
end Report_Error;
```

This generalized component is depicted in Figure 3b. This might constitute a domain-independent version of the reporting routine, according to our domain experts, although the only way to be certain that a component is compatible with all domains is to ensure that it does not depend on any other components. In Ada any such dependencies are revealed by the context clause. A later transformation will eliminate the dependence on Text_Io.

As noted, Ada affords us an opportunity to restore the assumption of using the specific string "Error: " and the standard output device through the use of default parameters without reducing the generality. This is shown below.

```
with Text_Io;
procedure Report_Error
        (Message : String;
         Intro : String := "Error: ";
         On_Device : Text_Io.File_Type :=
             Text_Io.Standard_Output) is
begin
  Text_Io.Put_Line (On_Device, Intro & Message);
end Report_Error;
```

At this point, two details remain (the use of

Text_Io.Put_Line and the prepending of a user string). The use of Put_Line could be removed through tailoring (below) but the removal of the choice to concatenate an introductory string could not be done within the language. For that degree of flexibility, generated reuse would be required. Once a generalization is needed which is not language-supported, the costs are considerably higher. One way to reduce those costs is to provide tool support for the generalization, a process which amounts to establishing a new language to accomplish the generlization. The MIS system described in [AIC] has reduced their software generation costs in this fashion.

This points out the obvious conclusion that the cost of a generalization depends on the level of language or tool support for it. One way to estimate cost is to begin with an ordinal scale of difficulty and then to move to a more detailed scale after more analysis has been done. For example, it was noted that value parameterization is relatively straightforward. This would be at the lowest end of an ordinal effort scale. Above that would be tailoring parameterization such as Ada's generic formal type and subprogram parameters. At the hardest end of the scale would be software generation, with tool-supported generation being easier than custom-built generation. A more detailed approach to effort would be to relate the cost to the number of lines of code that must be written, changed, or added.

It can require a judgment call to choose what details to remove and what function to leave in the component. For example, in the above example, the fact that the original literal string was broken up into a standard prefix and a user-supplied message was only one possibility for generalization. One guideline is to leave operational parts of a component intact and to allow the operands to be supplied by the reuser. A discussion of the separation of operations from operands can be found in [Bailey and Basili].

The simple error-reporting example from before can also be re-engineered into a tailored component using the Ada language. The difference between this result and the layered result is that the reusers will have to perform slightly more work in order to instantiate the component, but then subsequent calls can be simpler. As suggested, tailoring in Ada through the use of generics is seen as a harder process than value parameterization but easier than software generation. A tailored example of the component follows.

```
with Text_Io;
generic
  Intro : String := "Error: ";
  On_Device : Text_Io.File_Type := Text_Io.Current_Output;
procedure Report_Error (Message : String);

procedure Report_Error (Message : String) is
begin
  Text_Io.Put_Line (On_Device, Intro & Message);
end Report_Error;
```

Unfortunately, this is illegal in Ada since a limited type (Text_Io.File_Type) is not permitted as a generic value parameter. This is an example of where strong static checking can be at cross purposes with generalization and reuse. If it were legal, nevertheless, the user would have the responsibility for providing the introductory string and the output device one time (at the time of the generic instantiation) thus tailoring the component for further reuse. From then on, the component would be no more difficult to use (from the standpoint of parameterization) than the original non-general version.

To avoid this limitation of generic parameters, a solution could be obtained by generating the specific component desired, using tools outside of the Ada language. The generated component could look exactly like the original component but the reusable software would no longer be considered the component itself, but rather the generator which creates it. In this case, the generator would emit a Report_Error procedure which was hard-coded to write the error message on a given device. The value of that device would be given as a parameter to the generator. More examples of generation are shown later.

A different tailoring would also be possible. As mentioned earlier, the dependence on Text_Io can be eliminated by requiring that the client tailor the component to use a particular string-processing routine. This makes the component completely independent, with the persistence of the use of a standard prefix as the only detail which is retained from the original version.

```
generic
  Intro : String := "Error: ";
  with procedure Put (S : String);
procedure Report_Error (Message : String);

procedure Report_Error (Message : String) is
begin
  Put (Intro & Message);
end Report_Error;
```

Note that this most general version is also the least functional. Nevertheless, the ability to tailor the component once within a program and to then use it with the same level of effort as the first layered transformation makes it of some value. The reuser has additional work to do with this solution, as well. For example, unless the error messages are to be written to standard output, the subprogram to be passed to the generic formal Put

procedure has to be written. This means that the effort to reuse a tailored component could be greater than the effort to reuse a component generator. So, the effort to generalize is not always proportional to the corresponding effort to reuse.

By examining existing systems and by observing the opportunities to generalize their parts according to these different methods of reuse, choices become available in the ways in which the software can be re-engineered for future reuse. The next section describes a simple mail system in terms of its conventional configuration as a custom-built application and then in terms of the various ways the parts of it can be generalized using the above methods.

Re-Engineering a Simple Electronic Mail System

This section takes a simple electronic mail system through transformations to yield components which can be combined using the three methods described above. In the interests of space, parts of the examples and some identifier names have been abbreviated, and no bodies are shown. Complete listings of the examples are available from the authors.

In a conventional design, one component, or package, of a mail system could be used to manage the mailboxes of the users and a second could manage the messages, or the constituents of a mailbox. This would represent a conventional encapsulated or "object-based" design of the system where the mailbox package would allow operations such as create, add a message, delete a message, return a message, and perhaps displaying a directory of messages, maintaining the status of each message, and so on. The message package would allow message creation and display, and possibly reply construction, forwarding, etc.

In a typical arrangement, using either Ada or an object-oriented language such as Smalltalk, the mailbox package (or object) would depend upon the message package to obtain the use of the declaration of message objects, in order to arrange those objects into mailboxes. In Ada, the specifications for each of these two packages might reasonably be:

```
package Messages is
   type Username is ...
   type Line is ...
   type Text is ...
   type Message is private;
   procedure Set_Sender (M : in out Message; To : Username);
   procedure Set_Receiver (M : in out Message; To : Username);
   procedure Set_Subject (M : in out Message; To : Line);
   procedure Set_Body (M : in out Message; To : Text);
   function Sender_Of (Msg : Message) return Username;
   function Receiver_Of (Msg : Message) return Username;
   function Subject_Of (Msg : Message) return Line;
   function Body_Of (Msg : Message) return Text;
private
   type Message is
      record
         Sender : Username;
         Receiver : Username;
         Subject : Line;
         Msg_Body : Text;
      end record;
end Messages;

with Messages;
package Mailboxes is
   type Message is new Messages.Message;
   -- derive an equivalent type Message
   Max_Mailbox_Size : Natural := 1000;
   subtype Box_Size is Natural range 0 .. Max_Mailbox_Size;
   type Mailbox (Size : Box_Size := 0) is private;
   procedure Store (Box : Mailbox; Owner : String);
   procedure Retrieve (Box : in out Mailbox; Owner : String);
   function Size (Of_Box : Mailbox) return Box_Size;
   function Msg_At (Position : Natural; In_Box : Mailbox)
       return Message;
   procedure Remove (Num : Positive; In_Box : in out Mailbox);
   procedure Append (Msg : Message; To_Box : in out Mailbox);
   procedure Mark_Read (N : Natural; In_Box : in out Mailbox);
   procedure Mark_Unread  ...
   procedure Mark_Answered ...
   procedure Mark_Deleted  ...
   procedure Mark_Undeleted ...
   function Is_Read
      (Msg_Number : Natural; In_Box : Mailbox) return Boolean;
   function Is_Answered ...
   function Is_Deleted  ...
   No_Msg_At_Position : exception;
private
   type Attributes is (Deleted, Read, Answered);
   type Attr_Sets is array (Attributes) of Boolean;
   type Mail_Item is
      record
         Item : Message;
         Status : Attr_Sets;
      end record;
   type Item_Array is array (Positive range <>) of Mail_Item;
   type Mailbox (Size : Box_Size := 0) is
      record
         Items : Item_Array (1 .. Size);
      end record;
end Mailboxes;
```

These packages are depicted in Figure 4a. As shown, the Messages package is an example of an independently reusable layer, and the Mailboxes package constitutes a layer on top of the Messages package. (Since the constituent types of Username, Line, and Text are not shown, it might be the case that they would be comprised of user-defined types, making the Messages package dependent on other client software.) Realizing that the decision of how to implement the constituents of a message represents one of the opportunities for generalization of this package, the components of a message could be supplied as parameters to a generic version of this package. This would constitute a tailored version of the package:

10005788L

```
generic
  type Username is private;
  type Line is private;
  type Text is private;
package Gen_Messages is
  type Message is private;
  ... -- as before
end Gen_Messages;
```

This generalization is shown in the top part of Figure 4b. The effort to perform this tailored generalization is in line with other tailoring efforts discussed in the previous section. The declaration of three generic formal parameters is one measure of the work performed. Also, the reuse effort implies the declaration of actual type parameters to be associated with these generic formal types. One way to quantify the effort to generalize, then, is to claim that three declarations are required. Three declarations are also required of the client reuser.

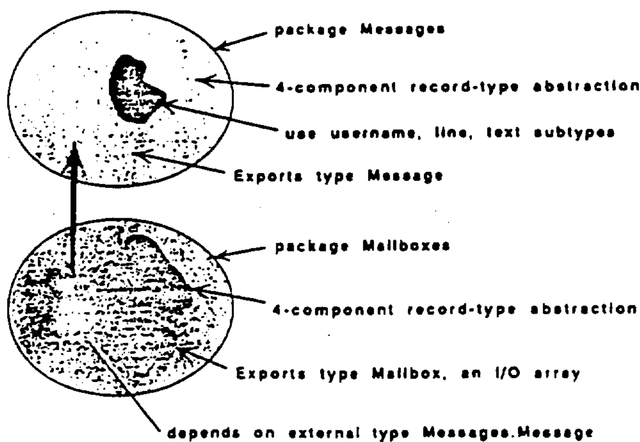

Figure 4a.

Using the conventions shown previously, this depicts the process of tailoring the Messages and Mailboxes packages from the text.

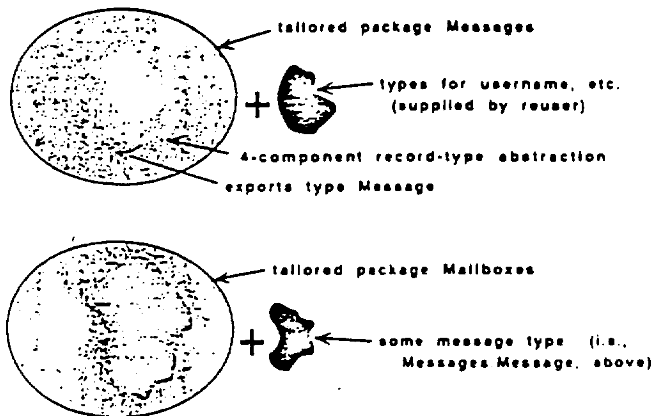

Figure 4b

The specific component types of a Message have been removed as well as the dependency of Mailboxes on Messages. The reuser will re-establish this link

Going beyond this somewhat tailored version, notice that even the structure of a message could be a candidate generalization. In this case, tailoring would be difficult or impossible within the confines of the Ada language so generation is required. Generation is feasible since the contents of the Messages package could be deterministically described if one were to specify the constituent components of a message. For example, if no subject line were wanted, the original package could instead have been written:

```
package Messages is
  type Username is ...
  type Text is ...
  type Message is private;
  -- procedures Set_Sender, Set_Receiver, Set_Body
  -- functions Sender_Of, Receiver_Of, Body_Of
private
  type Message is          -- no Subject component
    record
      Sender : Username;
      Receiver : Username;
      Msg_Body : Text;
    end record;
end Messages;
```

Or, if a message with a date and time stamp were desired, the abstraction could be augmented with an additional component, such as with the standard type Calendar.Time:

```
with Calendar;
package Messages is
  type Username is ...
  type Line is ...
  type Text is ...
  type Message is private;
  -- procedures Set_Sender, Set_Receiver, Set_Body,
  --        Set_Subject, and Set_Time
  -- functions Sender_Of, Receiver_Of, Body_Of,
  --        Subject_Of, Time_Of
private
  type Message is
    record
      Sender : Username;
      Receiver : Username;
      Time_Stamp : Calendar.Time;    -- new
      Subject : Line;
      Msg_Body : Text;
    end record;
end Messages;
```

Although the generic feature in Ada is not powerful enough to allow these variations as tailoring of a single common package, all of the Message package examples (as well as their corresponding bodies) could have been generated automatically, given the desired set of components for objects of type Message. This, therefore, becomes an example of generated reuse, where the generator is the reusable software and not the actual message package software. For example, a simple editor-substitution generator has been constructed which accepts input such as

the following and emits Ada equivalent to the example shown above.

```
Generate_Package
    (Context => "",
     Local_Decls =>
       "subtype username is string(1..10);" &
       "subtype line is string(1..60);" &
       "subtype text is string(1..80);",
     Package_Name => "messages",
     Private_Type => "message",
     Set_1 => "set_sender",
     Set_2 => "set_receiver",
     Set_3 => "set_subject",
     Set_4 => "set_body",
     Get_1 => "sender_of",
     Get_2 => "receiver_of",
     Get_3 => "subject_of",
     Get_4 => "body_of",
     Local_Type_1 => "username",
     Local_Type_2 => "username",
     Local_Type_3 => "line",
     Local_Type_4 => "text");
```

The effort to construct this generalization amounted to the writing of about 20 lines of software and the building of templates from the original unit. The effort to reuse the component is the construction of the above call. This could be seen as effort equivalent to declaring 17 string constants.

Note that, at this level of generality, which came at considerably higher cost than the previous tailoring, more than just a message package for a mail system could be generated. Any private type implemented as a record of components with set procedures and access functions could be generated with such a program. Therefore, this represents a domain-independent form of the component, where any mail system details are supplied by the reuser. So, the benefit of applying this substantial generalization effort is that the component can now be used by many domains. In fact, we will see that this same generator can be used to replace part of the Mailbox package, as well.

Although the style of the Mailbox package is not as general as the Messages package, there are several opportunities to make it more general and therefore more reusable in other contexts. For example, it could be tailored by making the constituent type Message and the maximum mailbox size generic formal parameters:

```
generic
  type Message is private;
  Max_Mailbox_Size : Natural := 1000;
package General_Mailboxes is
  ... -- same as package Mailboxes, above
end General_Mailboxes;
```

This arrangement of the Mailboxes package is shown in the bottom part of Figure 4b. Fortunately, no operations on the type Message were needed by the package Mailboxes, otherwise those operations would have had to have been passed as generic parameters.* Therefore, following the convention suggested above, the generalization effort here is the effort to write two generic formal parameter declarations. Reuser effort is the choice of a type and a value to perform the instantiation.

Beyond the relatively simple generalization shown above, it can be observed that the Mailbox abstraction is actually composed of a four-component record-type abstraction and an array. Reusing the previously described example of private record type abstractions, the package Mailboxes could be divided into two separate abstractions as follows:

```
generic
  type Message is private;
package General_Mail_Items is
  type Mail_Item is private;
  procedure Set_Message
    (An_Item : in out Mail_Item; To : Message);
  procedure Set_Read
    (An_Item : in out Mail_Item; To : Boolean);
  procedure Set_Answered ...
  procedure Set_Deleted ...
  function Get_Message (An_Item : Mail_Item) return Message;
  function Is_Read (An_Item : Mail_Item) return Boolean;
  function Is_Answered (An_Item : Mail_Item) return Boolean;
  function Is_Deleted (An_Item : Mail_Item) return Boolean;
private
  type Mail_Item is     -- a modified implementation
    record
      Item : Message;
      Read : Boolean;
      Answered : Boolean;
      Deleted : Boolean;
    end record;
end General_Mail_Items;

generic
  type Mail_Item is private;
  Max_Mailbox_Size : Natural := 1000;
package General_Mailboxes is
  subtype Box_Size is Natural range 0 .. Max_Mailbox_Size;
  type Item_Array is array (Positive range <>) of Mail_Item;
  type Mailbox (Size : Box_Size := 0) is
    record
      Items : Item_Array (1 .. Size);
    end record;
```

*If Ada supported full inheritance, it would be possible to write the Mailbox abstraction so that it relies on certain operations to be defined for the generic formal type Message. The user would then guarantee that any expected functions would be available for any actual type parameter associated with the formal type Message, eliminating the syntactic complexity of passing them via additional generic formal subprograms. This illustrates one of the advantages of late binding, something that Ada disallows in order to ensure that required operations are available prior to the compilation of any instantiations of the generic.

```
procedure Store (Box : Mailbox; Owner : String);
  procedure Retrieve (Box : in out Mailbox; Owner : String);
  function Size (Of_Box : Mailbox) return Box_Size;
  procedure Remove
    (Mail_Item_At : Positive; In_Box : in out Mailbox);
  procedure Append
    (A_Mail_Msg : Mail_Item; To_Box : in out Mailbox);
  No_Msg_At_Position : exception;
end General_Mailboxes;
```

These packages are depicted by Figures 5b and 5c. In the above case, the client could obtain the functional equivalent to the original mailbox package via the following instantiations:

```
package Mail_Items is
     new General_Mail_Items (Messages.Message);
package Mailboxes is
     new General_Mailboxes (Mail_Items.Mail_Item);
```

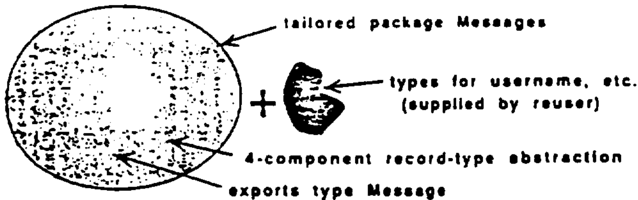

Figure 5a.

No additional changes are made during the second pass at tailoring the two packages. Only by generating the Messages package can the decisions about the structure of the abstract data type be generalized, since such a run-time tailoring is not possible within the Ada language.

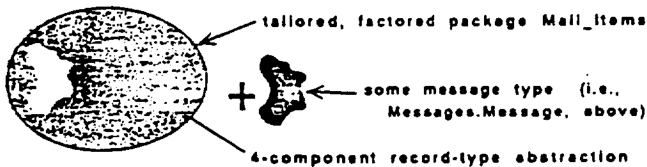

Figure 5b.

The Mailboxes package is broken into two components, one which implements Mail_Items as a record-type data abstraction, above.

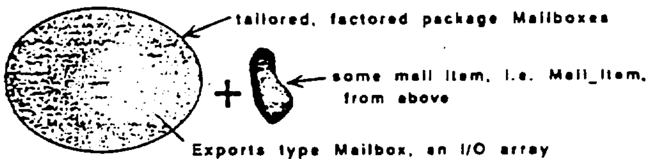

Figure 5c

The other package factored from the original Mailboxes package implements an I/O list of mail items. This no longer contains any problem-specific function other than implement lists, so it can be replaced with a general-purpose list abstraction, as shown in the text.

Two tradeoffs in this example are observed. First, the specific way in which package Mail_Item was structured originally was modified into the more general multi-component record shown here. This tradeoff was accepted in order to allow this implementation of Mail_Items to be similar to the implementation of Messages, which was previously shown to be highly generalizable. This is an example of how standardization limits the choices available to the implementer while increasing the generality of the resulting programs. For example, by adopting this approach, the generator program mentioned before could be used to generate an equivalent package to Mail_Items through the following input, thereby allowing the generation of both the Messages package and the Mail_Items package from the same reusable component:

```
Generate_Package
        (Context => "with messages;",
        Local_Decls =>
         "type message is new messages.message;",
        Package_Name => "mail_items",
        Private_Type => "mail_item",
        Set_1 => "set_message",
        Set_2 => "set_read",
        Set_3 => "set_answered",
        Set_4 => "set_deleted",
        Get_1 => "get_message",
        Get_2 => "is_read",
        Get_3 => "is_answered",
        Get_4 => "is_deleted",
        Local_Type_1 => "message",
        Local_Type_2 => "boolean",
        Local_Type_3 => "boolean",
        Local_Type_4 => "boolean");
```

The second tradeoff was to make the type Mailbox visible. This was necessary since the client software will have to gain direct access to a Mail_Item within a mailbox array in order to perform the operations from package Mail_Items on it. Simply returning a value of Mail_Item via a function call would not allow the user to set the components of a Mail_Item in a mailbox. An alternative solution would have been to implement the items in a mailbox as access values, each designating a Mail_Item. In this way, a function returning an access value would provide the capability for the client to modify the designated object, a Mail_Item. This situations occurs frequently when factoring composite abstractions into their constituent abstractions, and suggests that by presenting objects directly on the interface to an abstraction, rather than just their values, an abstraction can be made more general and reusable.

Further generalizations are not shown in detail in the interests of space. However, note that the above General_Mailboxes abstraction is the only remaining custom-made application code in the example. It amounts

to an ordered list of items of discernible size, to which items can be appended and from which items can be deleted, and which can be stored to and retrieved from files. Except for the ability to store and retrieve the lists, such an abstraction would probably be available in a library of generic data structures. Assuming the constituent objects are private and not limited private, it would be possible to perform binary input/output on them. So, it is not unreasonable to augment an existing generic abstraction to include storage and retrieval. Such an augmentation of a list resource could be accomplished by layering something like the following onto it.

```
-- Layering on a list abstraction:
with Simple_Lists;
generic
   type Item is private;
   type Item_Access is access Item;
package General_Mailboxes is
   package Item_Lists is new Simple_Lists (Item, Item_Access);
   type Mailbox is new Item_Lists.List;
   procedure Store (A_Box : Mailbox; To_File : String);
   procedure Retrieve
      (A_Box : in out Mailbox; From_File : String);
end General_Mailboxes;
```

To obtain the equivalent functionality as was provided by instances of the earlier package General_Mailboxes, the following declarations would now be required:

```
package Mail_Items is
        new General_Mail_Items (Messages.Message); -- same
   type Mail_Item_Access is access Mail_Items.Mail_Item;
   package Mailboxes is new General_Mailboxes
               (Item => Mail_Items.Mail_Item,
                Item_Access => Mail_Item_Access);
```

The client can treat the above package Mailboxes similarly to the earlier version; it will have all the same operations due to the derivability of those already implemented by Simple_Lists. Also, note that the mailbox implementation has been made private again by using designated objects to hold mail items. This would allow an Item_At function to return an access value to the actual mail_item and not just the value of that mail_item. This allows updates of the item via the operations that were defined in the Mail_Item package (Set_Message, Set_Deleted, etc.).

Measurement Summary

Measurement is required at two points of the software cycle. When candidate units are being identified and domain-specific details are being distinguished from problem-specific details, estimates of the generalization effort necessary to remove any give detail are required. At the time of reuse, estimates of the configuration effort necessary to adapt a component for reuse are required. Observations from conducting several generalizations have shown that an initial estimate based on an ordinal scale is possible. This scale has value parameterization as the easiest to perform for both generalization and reuse. Harder than this is type or operation parameterization, which requires tailored generalization in the case of Ada. The hardest form of generalization is building a special-purpose component generator. This can be made easier through the use of code-generation support tools.

After an initial evaluation of the generalization effort has been made and an approach to generalization has been determined, a more accurate assessment of the effort may be possible. The most direct indicator of the effort required is the number of lines of code that have to be written, changed or added. In many cases, a generalization can be accomplished with just a few lines of new or changed code. However, in the case of unsupported component generation, the entire generator may have to be written.

Reuse effort is easier to quantify since the component in question is already known. The effort to configure a generator or to instantiate a generic can be estimated based on the number of inputs or parameters required. In most cases, the usage of a tailored or generated component is similar regardless of whether the component was developed from scratch or obtained from a repository. However, even this step can be complicated by the fact that a development might choose to be constrained in some way in order to take advantage of an available component. The costs of such a decision can be especially difficult to estimate. In the long run, however, it is expected that the adoption of a component, similar to the adoption of a standard, is a cost-effective choice.

Another measure that is needed is an estimate of the future value of a unit in a repository. It may not be the best approach to populate a repository with many units which were inexpensive to generalize if they will rarely be needed. It would be better to spend the time performing a difficult generalization if the resulting unit will more than return that investment. Here again, domain experts will have to assist in making this determination.

Future Work

Progress is needed on metrics to quantify generalization and reuse effort. Effective metrics will open the way to establishing an economic model of reuse that could enable an organization to choose its optimal approach to reuse engineering. Note that the same approach or even the same specific model would not necessarily be best for two different organizations. One obvious reason for this is that one organization may concentrate in a single application domain while another organization may do work in many

10005788L

domains with very little repetition. The first organization may find its optimal approach to reuse is to develop a mature repository of domain-specific components while the second organization may find that only domain-independent components are likely to be cost effective.

In addition to the costs of generalization and reuse, an economic view of the software cycle suggested in this paper would have to deal with repository maintenance, component retrieval, component probabilities of reuse and cost savings, and the effort required of domain experts and repository experts. Current progress is being made in some of these areas by interviewing experts at one branch of the NASA Goddard Space Flight Center where reuse has been practiced for many years, originally with Fortran and more recently with Ada. The results of these interviews will assist us in formulating a more quantifiable model of the costs and benefits of reuse at that organization. It is hoped that this experience can then be extrapolated into a broader model of reuse engineering that can be adapted for use at other organizations.

References

[Bailey and Basili] J. Bailey and V. Basili, "Software reclamation: Improving Post-Development Reusability," in Proceedings Eighth Annual Conference on Ada Technology, Atlanta, Ga., 1990.

[Basili and Caldiera] V.R. Basili and G. Caldiera, "A Reference Architecture for the Component Factory," *Computer Science Technical Report Series*, University of Maryland, College Park, MD, March 1991, UMIACS-TR-91-24 or CS-TR-2607.

[Basili and Rombach] V.R. Basili and H.D. Rombach, "Support for Comprehensive Reuse," Software Engineering Journal, July 1991, (also, *Computer Science Technical Report Series*, University of Maryland, College Park, MD, February 1991, CS-TR-2606 or UMIACS-TR-91-23).

[Caldiera and Basili] G. Caldiera and V.R. Basili, "Identifying and Qualifying Reusable Software Components," *IEEE Computer*, Vol.24, No.2, Feb.1991, pp.61-70.

[Ellis and Stroustrup] M. Ellis and B. Stroustrup, "The Annotated C++ Reference Manual," Addison Wesley, 1990, p. 341.

[AIC] Ada Information Clearinghouse. "STANFINS-R - COBOL and C Programmers Moving Successfully to Ada." *Ada Information Clearinghouse Newsletter 8*, 2, June 1990.

John W. Bailey is a Ph.D. candidate at the University of Maryland Computer Science Department. He has been consulting and teaching in the areas of Ada and software measurement for nine years, and is currently consulting to Rational. He has an M.S. in computer science from the University of Maryland, where he also earned a bachelor's and a master's degree in cello performance. He is a member of the ACM.

Victor R. Basili is a professor at the University of Maryland's Institute for Advanced Computer Studies and Computer Science Department. His research interests include measuring and evaluating software development and is a founder and principal of the Software Engineering Laboratory, a joint venture of NASA, the University of Maryland, and Computer Sciences Corporation. He received a 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. He is a fellow of the IEEE Computer Society.

10005788L