Making Reuse Cost-Effective

BRUCE H. BARNES, *National Science Foundation*
TERRY B. BOLLINGER, *Contel Technology Center*

◆*Software reuse must be recognized as having the same cost and risk features as any financial investment. This article suggests analytical approaches for making good reuse investments.*

There has been quite a bit of debate in the last few years on the merits of software reuse. One reason for this continuing debate has been a question of scope: Should software reuse be defined narrowly in terms of one or a few specific methods (such as libraries of scavenged parts), or should it be defined broadly so it includes widely differing methods and processes?

The question of scope is important, since it really asks whether the concept of software reuse provides any major insight into the software-development process. A narrow definition implies that reuse is simply another development technique that, like many other techniques, is helpful in some contexts and inappropriate in many others. A broad definition implies that reuse incorporates one or more general principles that should be recognized and addressed explicitly in the software-development process.

We believe that reuse is in fact one of the fundamental paradigms of development and that, until it is better understood, significant reductions in the cost of building large systems will simply not be possible. We base this assertion primarily on our belief that:

The defining characteristic of good reuse is not the reuse of software per se, but the reuse of human problem solving.

A SCARCE RESOURCE

By "human problem solving," we mean those nonrepetitive, nontrivial aspects of software development and maintenance that cannot easily be formalized or automated using current levels of expertise. Unlike many other hardware and software resources used in development,

human problem solving cannot readily be multiplied, multiplexed, accelerated, or enhanced. Problem solving thus shares many of the characteristics of precious metals in the materials-processing industry: It must be used judiciously, replaced by less expensive resources when possible, and recovered for further use whenever feasible.

Problem solving shares many characteristics of precious metals. It must be used judiciously, replaced by less expensive resources when possible, and recovered for further use whenever feasible.

If human problem solving is viewed as a scarce resource then the three techniques of judicious use, replacement, and recovery correspond fairly closely to project planning, automation, and reuse:

- Good planning reduces the loss of human problem solving by minimizing redundant and dead-end work, by enhancing communication of solutions among developers, and by helping development groups select environments that support worker productivity.
- Automation is the classic process of tool building, in which well-understood work activities such as the conversion of formulas into assembly code are replaced with less costly automated tools such as compilers.
- Reuse multiplies the effectiveness of human problem solving by ensuring that the extensive work or special knowledge used to solve specific development problems will be transferred to as many similar problems as possible. Reuse differs from judicious planning in that it can actually amplify available problem-solving resources, just as automation can amplify the effect of well-understood, formally defined work activities.

Broad-spectrum reuse. If the key feature of effective software reuse is the reuse of problem-solving skills, it follows that reuse should not be restricted to source code. Any work product that makes problem solutions readily accessible to later projects is a good candidate for reuse. (A work product is any explicit, physical result of a work activity in the development and maintenance process.)

Examples of potentially reusable work products are requirements specifications, designs, code modules, documentation, test data, and customized tools.

This idea of *broad-spectrum* reuse[1] is particularly important because it has the potential to reduce costs substantially. It can reduce costs because reusing an early work product can greatly increase the likelihood of reuse of later work products developed from it.

For example, although reusing code modules from a custom database system can certainly reduce costs, reusing the system's overall functional specification could lead to the reuse of the entire set of designs, code modules, documentation, test data, and associated user experience that were developed from that specification. The chances of cost-effective reuse are much higher, both because more work products are reused and because the effort needed to adapt and integrate those work products into a new environment is greatly diminished.

Curiously, informal reuse of early work products is actually very common, but it often is not recognized because it masquerades as code-level reuse. Informal reuse of early work products occurs primarily when highly experienced developers use their familiarity with the functionality and design of a code module set to adapt those modules to new, similar uses.[2]

This powerful form of reuse is feasible only when developers can use their familiarity with early work products to zero in on the code modules that were derived from those products. The fact that these early work products may reside entirely within the developers' memories does not diminish their importance. Indeed, such situations point out the importance of automated support for reuse early in the life cycle. You need only have those developers retire or quit to discover how inefficient true code-level reuse is by comparison!

Reuse and automation. Thinking of effective software reuse as problem-solving reuse provides a good general heuristic for judging a work product's reuse potential. For example, modules that solve difficult or complex problems (like hardware-driver modules in an operating system) are excellent reuse candidates because they incorporate a high level of problem-solving expertise that is very expensive to replicate.

In contrast, a set of Unix date-and-time routines that differ only in their output formats are generally poor reuse candidates. You can program such format variants easily in Unix by using a stream editor such as Sed to modify the output of the standard date-and-time function. This approach is very flexible, and it requires little problem-solving skill beyond specifying the desired output formats and familiarity with a standard Unix tool. Trying to anticipate all the possible variants of date-and-time formats would in effect place a costly reuse technique (building a large library of variants) in direct competition with an existing automated method (generating variants by directly specifying output formats).

This example leads to a general rule:

Reuse should complement automation, not compete with it.

Automation and reuse are complementary in that automation tries to transfer as much work as possible to the computer, while reuse tries to make the most efficient use of work activities that cannot be fully automated.

Automated problem solving may one day help reduce the need for the human variety, but until that day arrives, software reuse offers a practical, high-potential approach to stretching the critical development resource of human problem solving.

REUSE AS AN INVESTMENT

If reuse is a vital component of the development process, what then is the best approach to understanding it and increasing its efficiency? First and foremost, we must recognize that reuse has the same cost and risk characteristics as any financial investment. Figure 1 illustrates why reuse should be viewed as an investment and introduces the reuse-investment relation.

Reuse-investment relation. The left side of this relation, the producer side, represents all the investments made to increase reusability. Reuse investments are any costs that do not directly support the completion of an activity's primary development goals but are instead intended to make one or more work products of that activity easier to reuse. For example, labor hours devoted specifically to classifying and placing code components in a reuse library are a reuse investment, since those hours are intended primarily to benefit subsequent activities.

Reuse investments should not be confused with the costs of making software maintainable, since maintainability costs are an integral part of building deliverable products. Instead, the completion of maintenance investments is the starting point for reuse investments. This distinction is particularly important because maintenance technology overlaps extensively with reuse technology. The threshold between these two costs is best defined in terms of global objectives, rather than in terms of specific technologies.

The right side of the reuse-investment relation, the consumer side, shows the cost benefits accrued as a result of earlier reuse investments. For each activity that applies reuse, the benefit is simply a measure in dollars of how much the earlier investment helped (or hurt) the activity's effectiveness.

To calculate the reuse benefit, you must first estimate the activity's cost without reuse and compare that to its cost with reuse. For example, if an activity placed several reusable components in a library, a subsequent activity would estimate its reuse benefit by comparing total development costs without reusing those components to total development costs when the reusable components are fully exploited.

You find the *total reuse benefit* by estimating the reuse benefit for *all* subsequent activities that profit from the reuse investment, even if those activities occur in the distant future. It is vital that you include all activities that benefit from a reuse investment, because it is this total benefit that determines the maximum level of reuse investment that you can apply economically to a set of work products.

In short, reuse investment is cost effective only when

$$R < B$$

where R is the total reuse investment and B is the total cost benefits. B is in turn defined as

$$B = \sum_{i=1}^{n} b_i = \sum_{i=1}^{n} [e_i - c_i]$$

where b_i is the benefit from reuse investment for activity i, e_i is the estimated cost of activity i without exploiting reuse, c_i is the cost of activty i when reuse investment is exploited, and n is the number of activities affected by the investment.

If an early estimate of the total reuse benefit indicates that it will be very small, you should make only a very limited investment in reuse technologies (beyond those needed to meet maintenance requirements). If estimates indicate that the total benefit will be very great, you should make substantial investments in advanced reuse technologies.

In either case, the failure of a development group to acknowledge the constraints of the reuse-investment relation could be disastrous from a cost perspective. After all, of what use is a very impressive, very advanced suite of reusable software if no one ever gets around to reusing it?

Producers and consumers. In Figure 1, activities that work to increase the reusability of work products are called reuse producer activities and activities that seek to reduce costs through reuse of work products are called reuse consumer activities. We intentionally used terminology with strong commercial implications because the underlying processes behind software reuse and commercial software marketing are strikingly similar.

Vendors of commercial software achieve net cost benefits only when their products are purchased (reused) enough times to cover development costs, just as reuse producers achieve a net benefit only when their work products are reused enough times to cover investment costs. Most reuse producers (vendors) and reuse consumers (buyers) differ from their commercial counterparts primarily in that the transfer of products between them takes place within a single company or project, rather than across company or organiza-

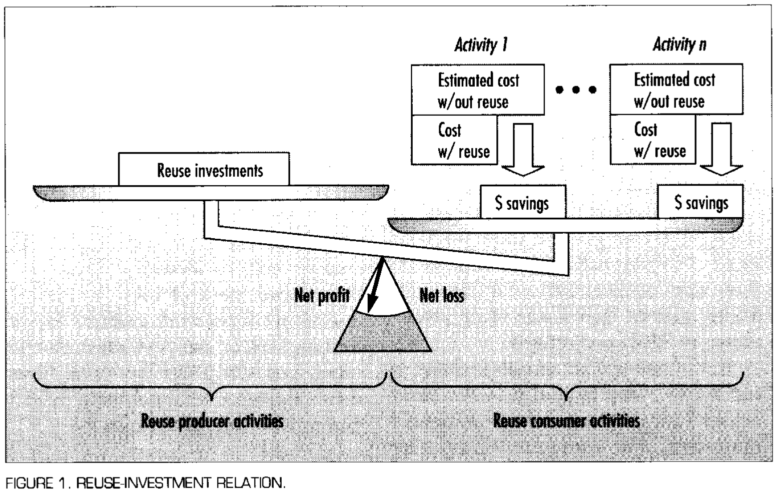
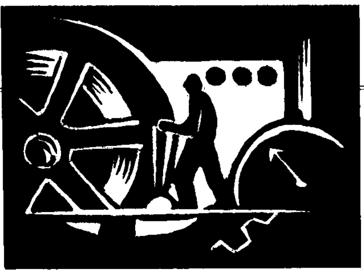

Activity 1
Activity n
Estimated cost w/out reuse
Estimated cost w/out reuse
Cost w/ reuse
Cost w/ reuse
Reuse investments
$ savings
$ savings
Net profit
Net loss
Reuse producer activities
Reuse consumer activities

FIGURE 1. REUSE-INVESTMENT RELATION.



tional boundaries.

The producer/consumer reuse model is rich in both managerial and technical implications. One obvious managerial implication is that organizations that fail to provide some form of payback incentives to producer projects are unlikely to succeed at reuse, since producers will, in effect, be penalized for making overhead expenditures that do not directly contribute to their development project.

Indeed, it is likely that one of the most significant inhibitors of reuse in the software industry is a lack of incentive strategies to encourage coordinated reuse investments. Without such incentives, reuse becomes a scavenger hunt, where each reuse customer must bear the full cost of finding, understanding, and modifying work products to meet his needs.

Scavenging is an extremely inefficient form of reuse because it duplicates reengineering costs each time the product is reused. Well-planned investments made up front by producers can make it much easier for consumers to find, understand, and customize the parts they need. Also, reuse producers can build in reusability, which avoids the significantly more costly task of reengineering reusability into existing work products.

MAKING REUSE COST-EFFECTIVE

Another way to express the reuse-investment relation is to define a *quality-of-investment measure*, Q, which is simply the ratio of reuse benefits to reuse investments:

$$Q = B/R$$

where R is the total reuse investment and B is the total cost benefits resulting from reuse investment.

The Q measure is just another way of saying, "Try to get the most for your money." If Q is less than 1 for a reuse effort, there was a net financial loss; if Q is significantly greater than 1, the reuse investment provided good returns.

In the case of commercial products, Q can become very large due to the many reuses (sales) afforded by commercial marketing. The commercial-marketing comparison again points out an important issue: Reuse investments are most likely to pay off when they are applied to high-value work products. Applying expensive reuse technologies to low-value work products is not at all likely to produce reuse winners, although in some cases such a strategy may result in particularly spectacular failures.

If the key to making reuse cost-effective is to increase the Q measure, how is such an increase best accomplished? The variables in the reuse-investment relation suggest three major strategies for increasing Q:

- ♦ Increase the level of reuse.
- ♦ Reduce the average cost of reuse.
- ♦ Reduce the investment needed to achieve a given reuse benefit.

We have developed ways to implement these strategies, and in the process we uncovered a few surprising implications about how reuse is closely linked to the general problem of developing high-quality software.

INCREASING REUSE

The first strategy for increasing Q is to increase the level of consumer reuse. However, the likely level of consumer reuse for a work product depends strongly on the product's intrinsic value. Components that incorporate a high level of unique expertise are far more likely to have a large reuse market than are weak products that lack distinctive features.

Increasing the number of actual instances of reuse thus should be viewed primarily as an analysis task, to identify the relative merit of investing in a set of work products. By performing such an analysis early in development, you can avoid low-value investments and more accurately determine the potential of high-value products.

One of the most significant inhibitors of reuse is a lack of incentive. Without incentives, reuse becomes a scavenger hunt, where each reuse customer must bear the full cost of finding, understanding, and modifying work products.

Reuse instances. A *reuse instance* occurs whenever a work product is actually reused in a subsequent development activity. Due to the strong (roughly linear) dependence of Q on the overall level of reuse, if you have low expectations for the number of reuse instances, you should keep the average level of reuse investment per unit of code low. But if you expect many reuse instances, the project may merit much more extensive reuse investments per unit of code.

In estimating reuse instances, an important first step is simply to ask questions. Often, if you ask the specifiers and architects of a new system to make an order-of-magnitude guess of the number of reuse instances (one? 10? 1,000?), they can do so with little difficulty. Even such rough approximations can help you avoid costly overinvestments.

Reuse and competition. The process of estimating reuse instances for a software component is linked to the much broader problem of understanding the commercial software market. This is because all forms of reuse are essentially competitive: Any group that wants to reuse software must decide whether to take that software from a corporate reuse library or buy it. If the quality of library software is low, if it requires extensive adaptation, or if it is poorly documented, it is entirely possible that a similar commercial product with an initially higher cost could in the long run prove to be less costly than the equivalent "free" software.

This means that reuse investors should be careful not to inflate instance estimates with reuse opportunities that are real but that are likely to be filled by products from other sources. As an extreme example, very few developers should seriously consider making a custom file system reusable, since this would place them in direct competition with a very advanced commercial market for database-management systems. A realistic estimate of the number of reuse instances in this case should be very low, perhaps zero.

You can greatly reduce the risks of mistakenly competing with external markets if you apply the principle that good software reuse is actually good problem-solving reuse. Most companies are particularly skilled in one or more problem-solving areas, which gives them important competitive advantages. A company that builds software for the National Aeronautics and Space Administration may have special expertise in flight-dynamics algorithms, for example. If a persistent need for this type of expertise is expected, the company might well find it profitable to build reusability into its flight-dynamics software. A rule of thumb summarizes such situations:

Build reusable parts for local expertise; buy reusable parts for outside expertise.

Variants analysis. Estimating reuse instances is the simplest form of a more general type of analysis that we call *variants analysis*. Variants analysis is to reusable software what requirements analysis is to traditional once-only software. Its objective is to quantify requirements for reuse investment up front, just as requirements analysis attempts to quantify requirements for functionality up front.

As in reuse-instance estimation, variants analysis in its simplest form consists simply of asking questions — explicitly examining how future development or maintenance efforts may use updated or altered versions of the current project's functional requirements.

More elaborate forms of variants analysis require a structured format to record such information. By analogy with requirements specifications, these repositories of information about likely future variants are called variants specifications.

Variants specifications. A variants specification is a requirements specification extended to include the best available information on how the activity's work products are likely to be reused.

To help designers translate these specifications into reusable products, they are stated in terms of product variants — functional variations of the primary product. Product variants can be described in many ways, ranging from explicit descriptions of multiple products to parameterized, generic requirements.

Unlike a requirements specification, a variants specification tries to describe an objective that is inherently heuristic — it must express a set of best guesses as to which of a potentially infinite range of product variants is most likely to be needed in the future.

This likelihood of use can be described in terms of a probability percentage, which may range from 100 percent for products that are specifically required as part of the product delivery, through moderate values (10 percent) for variants that are fairly good reuse candidates, to potentially very low values (0.01 percent) for product variants that are not likely to be needed but that could be very valuable if they are ever actually reused.

Creating a well-designed system is fundamentally a problem of anticipating its most likely variants and of applying design techniques that will ensure that the most likely variants will also be the least expensive to build during maintenance.

You can also use such likelihood-of-use percentages to set priorities, where a 100-percent rating would indicate a customer requirement that must be met to fulfill contractual obligations, while lower ratings would determine the relative value of variants.

Because variants specifications describe sets of software products, you can make them as simple or as elaborate as you want. If you describe only one product (all requirements have 100-percent priority), the variants specification is identical to a (simplistic) requirements specification. Better is a group of variants specifications that correspond to a typical requirements specification.

Although requirements specifications are not usually viewed in terms of reuse, they normally contain some variants requirements phrased in terms of modularity and portability. For example, a requirement that a product must be portable across Digital Equipment Corp., IBM, and Sun computers is a variants requirement, because it says in effect that there is a very high probability that the customer will reuse (port) the product to one or more environments.

Reuse, maintenance, and good design. Besides reusability, there is another intriguing reason explicit variants analysis could be a useful addition to the development process.

A key feature of a well-designed architecture is that it be easy to maintain, since such systems are more likely to have a long useful life span and less likely to be expensive to support. However, software maintenance is itself a form of reuse-with-replacement, in which new variants of entire systems are created and then substituted for the original systems.

From this perspective, the problem of creating a well-designed system is fundamentally a problem of anticipating its most likely variants and of applying design techniques that will ensure that the most likely variants will also be the least expensive to build during maintenance — in other words, variants analysis. Current design approaches apply general rules that, while they usually result in designs that support the necessary variants, lack the specificity provided by an explicit variants analysis.

If variants analysis is in fact a hidden dimension of good software design, then a better understanding of how to build highly reusable software could simultaneously lead to a more quantitative understanding of what makes a particular design "good" or "bad." It would also imply that good design may be a more relative concept than is generally recognized. A design that supports long-term maintenance very

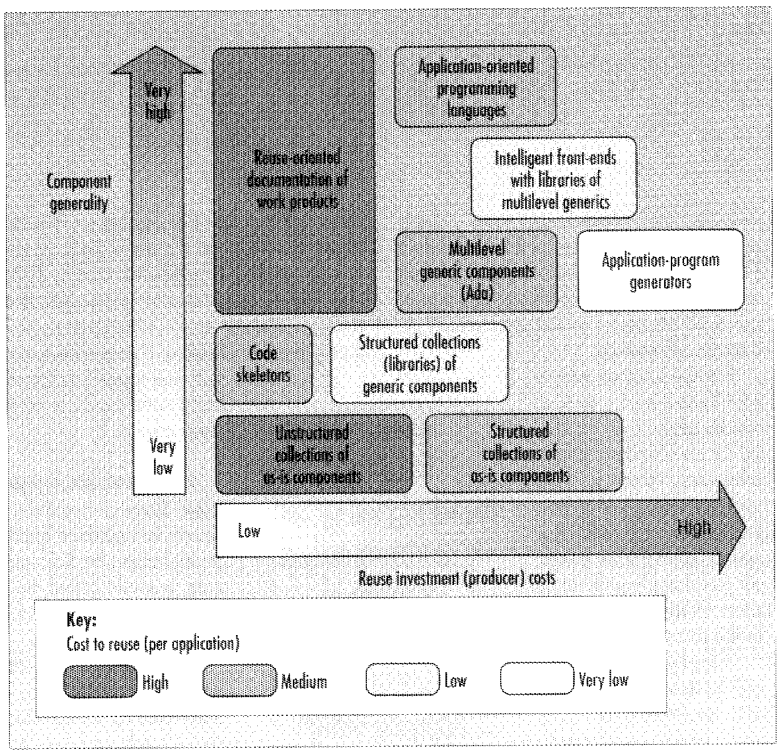

Component generality
Very high
Very low
Reuse-oriented documentation of work products
Application-oriented programming languages
Intelligent front-ends with libraries of multilevel generics
Multilevel generic components (Ada)
Application-program generators
Code skeletons
Structured collections (libraries) of generic components
Unstructured collections of as-is components
Structured collections of as-is components
Low
High
Reuse investment (producer) costs
Key:
Cost to reuse (per application)
High
Medium
Low
Very low

FIGURE 2. COST-BASED SELECTION OF REUSE TECHNOLOGIES. COMPONENT GENERALITY REFERS TO THE NUMBER OF READILY AVAILABLE VARIATIONS OF A PART. PRODUCER COST IS THE TOTAL COST OF MAKING PARTS READILY AVAILABLE TO LATER DEVELOPERS.

well in one corporate environment could simultaneously be badly mismatched to the long-term needs of a different corporate environment.

REDUCING COST

The second technique for increasing the investment-quality measure Q is to reduce the average cost of reusing work products by making them easy and inexpensive to reuse. Just as commercial vendors try to make their packages easy to adapt to the environments of many customers, reuse producers need to encourage reuse of their products by making them easy to find, adapt, and integrate into new systems.

How much a reuse producer can invest in such efforts will depend both on expected levels of reuse and the availability of appropriate reuse technologies. Reducing the cost of reuse thus depends to a large degree on selecting technologies that adequately support the needs of reuse customers while constraining reuse investment costs to acceptable levels.

Classifying reuse technologies by cost. To select reuse technologies on the basis of cost versus power, you first must compare them in these terms. Figure 2 shows some ballpark approximations of how various code-specific reuse technologies compare in terms of costs versus power.[3] (Comparable sets of reuse technologies for non-code work products such as designs and requirements specifications do not yet exist. This is a curious deficiency, since the development of such broad-spectrum reuse technologies may eventually prove to be a profitable enterprise.) The diagram has three dimensions: reuse investment costs, component generality, and cost to reuse.

Reuse investment costs. The reuse investment cost is the total cost to the producer to make parts readily available for reuse. Comparing technologies along this dimension alone will help eliminate major classes of reuse technologies as too costly. For example, program generators (exemplified by operating-system Sysgen programs) are very powerful and easy to use but require large investments.[4] You can justify such investments only if you anticipate many potential reuse instances, such as is true for operating-system installations.

Component generality. The component-generality dimension introduces the concept of the relative power of a reuse technology, expressed in terms of how many variations of a part can readily be obtained by applying that technology. Generality measures how well a particular technology can cover the needs of an application area. Because very few parts can be reused without some modification, a technology that provides large suites of easy-to-reuse, predefined variants greatly increases the total number of reuse opportunities.

But building generality into reusable parts tends to be expensive and labor-intensive. While Ada's generic procedures are undeniably more flexible and reusable than conventional Fortran-like procedures, they are also significantly harder to build.

Generative methods are an extreme example of component generality. In effect, these methods allow the synthesis of reusable parts from application-specific languages in much the same way that compilers allow the synthesis of object-code modules from higher level languages. Although generative methods provide high levels of generality and ease of use, they require extensive analysis and preparation. Their high investment cost thus makes them appropriate only when you anticipate very many reuse instances.

Cost to reuse. The cost to reuse is the total cost to the reuse consumer of finding, adapting, integrating, and testing a reusable component. In the simplest cases, you can minimize the cost to reuse by identifying a plausible level of reuse investment and choosing the corresponding reuse technology that has the lowest cost to reuse.

Thus, if a reuse producer group has identified the need for a moderate level of reuse investment and has adequately characterized the likely reuse instances, it might choose to use Ada generics, for example, to implement its reusable components.

The key phrase, however, is "adequately characterized." The problem is

that the generality of Ada generics is comparatively low, because they primarily substitute data types rather than modify functionality. If future reuse instances are poorly characterized, this means that highly specific generalizations could easily miss the target of actual future needs. Alternately, the total costs of generalization could become excessive as the developers try to use an overly restrictive technique to cover a very broad range of conceivable reuse instances.

REDUCING INVESTMENT COSTS

The prospect of overextending a reuse technology brings up the third strategy for increasing the *Q* measure: reducing investment costs. Just as a reuse technology that is not general enough can be unduly expensive for reuse consumers, a technology that is either overextended or more general than necessary can result in excessive costs to the reuse producer.

What you need is a way to analyze the information provided by variants analysis more carefully, so the level of generality provided by reuse technologies will match future expected needs.

The starting point for this matching is the concept of *instance spaces*, which let you define the generality level of reusable components more precisely. The instance space of a component or other work product is the full set of variants that can be retrieved at reasonable cost (defined as less than the equivalent full development cost) by a reuse consumer.

Figure 3 shows an instance space for an Ada generic component. An important feature of an instance space is that it makes no difference whether a specific instance of a part is actual (a part in a library) or virtual (a potential but as-yet-unrealized instantiation). Because they abstract out the issue of whether a part is physical, instance spaces let you compare highly diverse reuse technologies such as code libraries and generative methods.

Instance-space abstraction also takes manual methods into account, since they can be classified as having very large instance spaces (people can program nearly anything in time) but high consumer costs (programming is more expensive than, say, building new procedures with Ada generics).

Matching generality to needs. In variants analysis, you characterize consumer needs in terms of product variants. Characterizing variants this way lets you compare the results of variants analysis to the instance spaces of generalized components or work products.

Figure 4 shows such a comparison, including the three groups of instances that normally result from an intersection of variants and instance spaces. These three groups are labeled in terms of how they are perceived by reuse consumers: extensive-adaptation instances, moderate- to minimal-adaptation instances, and unused instances.

♦ Extensive-adaptation instances are actually accessed and used by consumers, but they are so difficult to adapt that the cost benefits are negligible or possibly even negative. These instances represent lost opportunities where a reusable part could have provided cost benefits, but its generalization failed to anticipate the needs of the consumer adequately.

♦ Moderate- to minimal-adaptation instances represent significant cost benefits to one or more customers. In these cases, the part was generalized so it was easy to retrieve and adapt.

♦ Unused instances are a large category that, while they should be expected in any application, should be controlled by reuse producers to prevent overinvestment. Many low-cost unused instances are

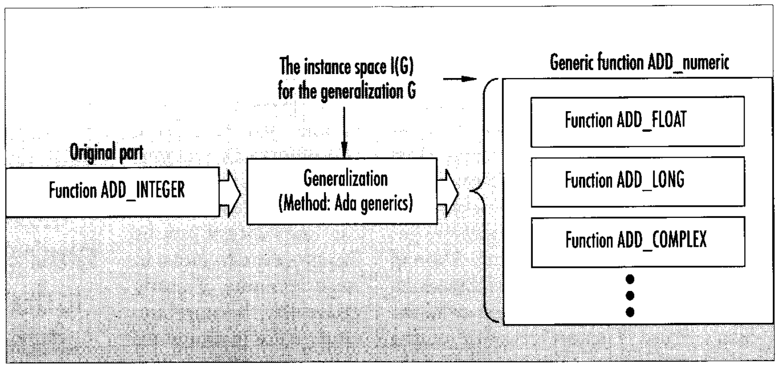

The instance space I(G) for the generalization G
Generic function ADD_numeric
Original part
Function ADD_INTEGER
Generalization (Method: Ada generics)
Function ADD_FLOAT
Function ADD_LONG
Function ADD_COMPLEX

FIGURE 3. INSTANCE SPACES.

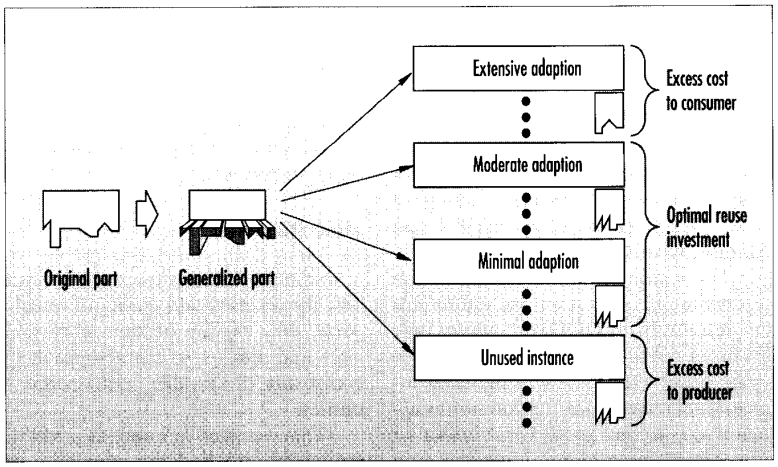


FIGURE 4. COST OBJECTIVES IN BUILDING HIGHLY REUSABLE SOFTWARE.

generally acceptable and even desirable as insurance, but many unused instances created with expensive technologies could easily result in reuse-investment losses.

Comparing instance spaces to variants analysis works best when the product is relatively small. Otherwise, the size of the instance space becomes so large that it becomes impractical to directly characterize the moderate- to minimal-adaptation instances.

Even for small components, the instance spaces are typically so large that they are better handled by specifying characteristics than by explicitly enumerating all possible instances. Thus, you would characterize the moderate- to minimal-adaptation instance space for the Ada generic example of Figure 3 by specifying key constraints, such as the potential need for any legal scalar variant of the routine, the exclusion of matrix-algebra variants, and the need for a particular range of computational accuracies.

Again, the most important reason to perform such characterizations is to encourage reuse investment decisions based explicitly on expected needs, instead of on habit or what's easiest to do. For example, if you could firmly establish that the generic addition routine in Figure 3 would never be reused in any applications requiring complex algebra, you could avoid the extra expense of generalizing that routine to include complex addition. Conversely, if the routine was part of a larger package for which conversion to matrix algebra was plausible, the extra effort to add hooks for such conversion might be worthwhile.

Mixing reuse technologies. You need not pick just one reuse technology for a set of components. In fact, it is more likely that the best and most economical coverage of consumer needs will be provided by two or more reuse technologies.

The reason for this is related to risk reduction. In most cases, you will be able to describe likely consumer variants only in broad terms. While you may be able to state firmly that there will be many instances of reuse within a broad set of variants, you may not know where in that set those instances will fall. Ada generics are a simple example of this situation, since they involve cases where the need for new data types is well established but the exact definition of those data types cannot be known until the consumer actually needs a new routine.

Such late-binding situations are best covered by reuse technologies that provide a high level of generality, but they are relatively costly to the consumer. An example of such a technology is reuse-oriented documentation, which amounts to simply ensuring that the results of variants analysis are embedded as documentation throughout a development effort's work products.

On the other hand, instances that are very good candidates for reuse should be handled using technologies that pass the lowest possible cost to the reuse consumer. Moving costs to the producer for such sure-bet cases is nearly always appropriate, since the developer of a system can usually generalize its work products far more cheaply than later reuse consumers can reengineer them.

In these cases of certain reusability, it may be inappropriate to invest in large amounts of costly generality, because the target reuse instances are already well characterized. An extreme example is a reuse variant that is fully specified and is 100-percent certain to be needed. In this case, the ideal reuse strategy would be to build the reuse instance at the same time the primary product is built.

Paradoxically, reuse-intensive development is best accomplished by focusing more on how to change software effectively than on how to keep it from changing.

REUSE STRATEGIES

Although instance spaces help represent the diversity and quality of reusable parts, they rapidly become intractably large and difficult to characterize as the complexity of reusable components increases.

What we need is a way to introduce modularity into the design of reusable systems so the instance-space size can be made more tractable. Modular reuse strategies should take a divide-and-conquer approach that supports both the design of new, highly reusable systems and the analysis of existing, potentially reusable systems.

The starting point is to recognize that you can view reuse (and development) as the construction of new systems by combining two forms of functionality:

- ♦ Invariant functionality, which is the set of components (or component fragments) that are used without change. Mathematical routines are common examples. By definition, invariant functionality alone cannot create a new system, since it lacks the customization needed to meet a new set of requirements.
- ♦ Variant functionality, which is the set of new functionality (software) that must be added to customize invariant components. Variant functionality may be as simple as a set of arguments passed to a parameterized package, or as complex as full, from-scratch, new-system development.

Mixing variant and invariant functionality. Regardless of how simple or complex a system is, it will always contain some mix of these two functionality types. Invariant functionality provides the kernel of functionality around which new systems are constructed; variant functionality provides the novel functionality that lets a system address a new set of needs.

The objective of reuse-intensive development is not to do away with the variant component, since that would, in effect, prevent any new needs from being addressed. The objective of effective reuse-intensive development is the creation of invariant components that help focus the development of variant functionality into very precise, succinct statements of the differences between the old and new systems.

Paradoxically, then, reuse-intensive

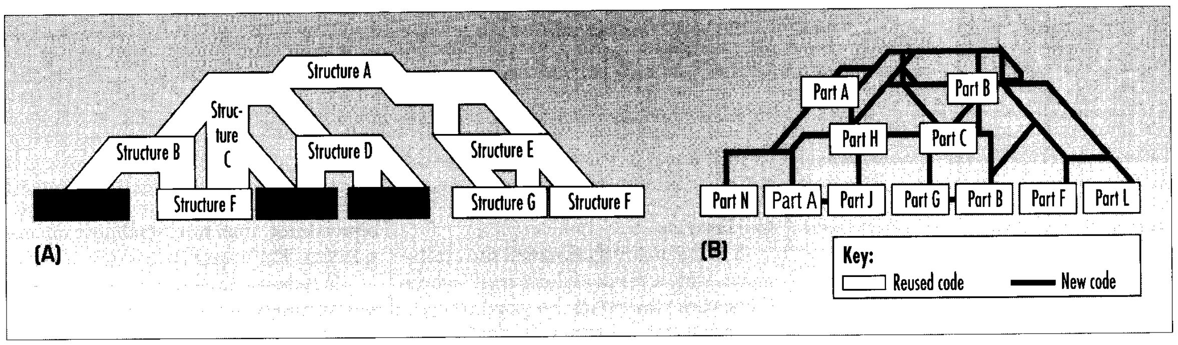


FIGURE 5. FUNDAMENTAL REUSE STRATEGIES. (A) ADAPTIVE REUSE; (B) COMPOSITIONAL REUSE.

development is best accomplished by focusing more on how to change software effectively than on how to keep it from changing. Well-focused mechanisms for expressing variant functionality will do far more to keep large sections of code invariant than will arbitrary attempts to build around existing code components.

The concept of variant and invariant functionality can readily be extended to noncode work products by analyzing how they can be built by combining baseline components of various sizes and types. As with executable code, the objective in building, say, a highly reusable system specification is to develop a set of baseline components that support concise, succinct descriptions of how new specifications differ from existing ones. Technologies such as hypertext are being applied toward this type of objective,[5] but far more work on characterizing the features of "good" variant and invariant components of noncode work products must be done.

Two fundamental strategies. If reusable systems are always constructed from some mix of invariant and variant functionality then the way in which they are combined is an important criterion for evaluating a system's reuse characteristics. This observation leads immediately to the definition of two broad, complementary reuse strategies, which are shown in Figure 5:

♦ Adaptive reuse uses large frame structures as invariants and restricts variability to low-level, isolated locations within the overall structure.[6] Examples include changing arguments to parameterized modules, replacing low-level I/O modules, and altering individual lines of code within modules. Adaptive reuse is similar to maintenance in that both try to isolate changes to minimize their system-wide impact.

♦ Compositional reuse uses small parts as invariants and variant functionality as the glue that links those parts.[7,8] Examples include constructing systems from parts in a reuse library and programming in high-level languages. As its name implies, compositional reuse is similar to conventional programming, in which individual functions of moderate complexity are composed according to some grammar to create new, more powerful functions.

Differences. In terms of cost and component-generality characteristics, adaptive and compositional reuse are strikingly different. For example, because adaptive reuse tries to keep most of the overall structure invariant, it tends to be application-specific and comparatively inflexible, but it helps keep both producer and consumer reuse costs under control.

By contrast, compositional reuse can be very flexible if the initial set of reusable components is sufficiently rich and generalized. However, constructing such a generalized component set (such as the functions in an application-specific language) can be very expensive for the reuse producer. Also, consumer costs for compositional reuse tend to rise rapidly as the complexity of the constructed software increases; when compositional reuse is extended too far, it begins to look more and more like conventional programming.

Coverage level. For compositional reuse, we make a further distinction based on the coverage level of part sets. Coverage level refers to the ability of a set of parts to address an application area without forcing the consumer to use a lower level language such as Ada or Cobol. There are two major coverage types:

♦ Full-coverage sets are sufficiently rich and complete to let you solve new problems within a well-defined application area using only those reusable parts. Examples of full-coverage sets include application-specific languages and abstract data types implemented with Ada packages.

♦ Partial-coverage sets provide representative parts that act as examples of the component types needed to solve problems in an application area. However, partial-coverage sets are not sufficiently generalized to let you solve all problems in that application area without resorting to the use of lower level languages. A library is a good example of a partial-coverage set, because the parts in it usually are neither highly generalized nor complete in their coverage of a problem area.

Partial-coverage sets are likely to be much less expensive to produce, but they are also more likely to be expensive to reuse because the parts they contain are hard to understand, modify, and test.

Combining strategies. By themselves, neither adaptive nor compositional reuse strategies are general enough to cover the full range of potentially reusable structures. However, the structured combination of these two approaches creates hybrid strategies much broader in coverage. Figure 6 shows the two major hybrid approaches.

♦ Full-coverage hybrid reuse uses an adaptive framework to keep overall costs down and "bubbles" of full-coverage compositional sets at key locations to provide flexibility.

Unix's Termcap package demonstrates this concept nicely, even though it was not designed as a reuse technology. With Termcap, you build drivers for new terminal types using interface descriptions written in the special-purpose Termcap language. These interface definitions are actually examples of compositional reuse, since the Termcap language allows ready access to (reuse of) a complex suite of general-purpose driver routines. Because you can handle hardware variations with a specialized minilanguage that strongly encapsu-

lates such variations, you can transfer (adaptively reuse) higher level programs that use Termcap among systems with few or no changes.

♦ Partial-coverage hybrid reuse is very similar to the full-coverage hybrid, but it allows the bubbles to be either full-coverage or partial-coverage — they need not be fully generalized for handling the problem area they address.

Many types of maintenance are equivalent to partial-coverage hybrid reuse, since they are based on localized modification of components that have never been fully generalized.

Cutting costs. Partial-coverage hybrid reuse provides the best overall framework for defining the divide-and-conquer approach that is our motive for exploring reuse strategies. Partial-coverage hybrid reuse keeps the overall structure invariant, so producers can focus on smaller, more precisely defined problem areas. Reuse investments in those problem areas then can produce minilanguage sets that permit the flexibility consumers need.

When variants analysis indicates a need for extensive generalization (Termcap's terminal-interface problem, for example), producers can invest in expensive technologies such as application-specific languages or program generators. When needs are clearly identified but full generalizations are not justified due to poor characterizations or limited numbers of expected reuse instances, producers can provide partial-coverage sets such as localized libraries.

It is the ability to combine such techniques within the partial-coverage hybrid framework that makes it a good strategy for keeping producer costs in check.

REUSE AND PARAMETERIZATION

When we say that highly generalized parts define minilanguages for solving specialized problems, we don't mean to imply that these languages are true general-purpose languages — they are not. But designers use them in a way similar to general-purpose languages. Thus, sets of reusable graphics routines are a minilanguage for constructing and modifying diagrams; libraries of mathematical routines are minilanguages for dealing with mathematical problems.

From this perspective, good sets of reusable parts should possess the same general characteristics of expressive power and ease of composition that are the hallmarks of good programming languages.

Producers and consumers can further reduce reuse costs by designing appropriate, well-structured minilanguages. While this language design is identical to the task of generalizing parts, the language-design perspective places a much stronger emphasis on the integration of parts. The minilanguages that result should define clear and succinct problem solutions through the composition of a relatively small number of objects and operators.

The design of optimal minilanguages can be assisted by recognizing a curious equivalence that has significant consequences. The equivalence is this:

All parameterizations can be interpreted as minilanguages, and all minilanguages can be interpreted as parameterizations.

Parameters as programs. At first glance, the statement that parameterizations and languages are in some way equivalent seems outlandish, particularly if it encompasses data parameterizations. After all, what does the passing of two complex numbers to a Fortran subroutine for complex division have to do with expressing a problem in a minilanguage?

Actually, quite a bit. A Fortran complex-division subroutine is a formal system for solving problems in a very small, very precisely defined domain: division of complex numbers. There are only two terms in this domain's minilanguage, which may be paraphrased as "define the dividend to be (value)" and "define the divisor to be (value)."

These terms are "coded" with a simple positional notation, but the semantic interpretation implied by the paraphrased versions most definitely must be met to obtain valid results.

Ironically, such routines are not usually thought of as defining small languages precisely because of their effectiveness: They are very succinct and directly address the key variability (new data values) needed to solve problems in their domains. In Ada, you can make this equivalence more obvious through the use of named parameters that help preserve such

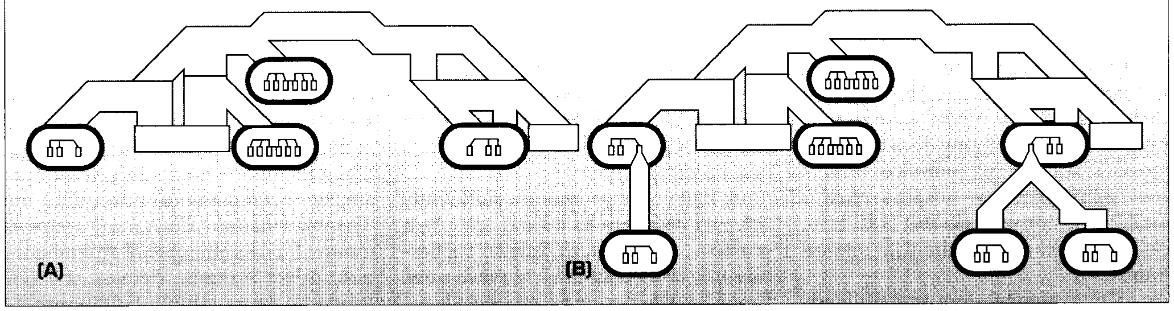


FIGURE 6. HYBRID REUSE STRATEGIES.(A) FULL-COVERAGE HYBRID; (B) PARTIAL-COVERAGE HYBRID.

semantic implications.

Programs as parameters. Looking at the equivalence from the other side, you can view a compiler as a formal system for interpreting program parameter values. Like any other parameter, a program consists of a string of binary data that has a specific meaning when passed to the formal system (compiler) for which it was constructed.

The key way in which a program differs from a traditional set of parameter values is that instead of defining a problem solution in terms of many relatively independent terms, a program uses comparatively few highly interrelated terms.

There is no rigid boundary between low-complexity schemes in which parameters are fairly independent and high-complexity cases in which they are highly interrelated. For example, some application-specific languages such as database-report generators often provide ways to specify common "programs" (reports) with very simple, parameter-like specifications.

Also, conventional parameterization schemes may include language-like features to reduce the number of necessary parameters. For example, many Unix tools such as Grep and Sed use parameterizations that range in complexity from the passing of simple flags for commonly needed variants of their functionality to the passing of complicated, program-like character strings for customizing their behavior in very specific ways.

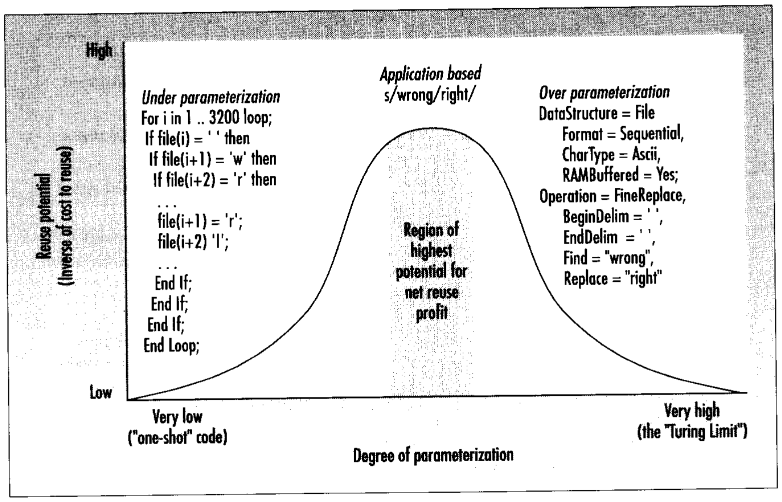
Limits of parameterization. Figure 7 shows that extreme parameterization does not necessarily lead to a more understandable or reusable system, since extreme parameterization is functionally equivalent to developing a highly generalized minilanguage.

Very high levels of parameterization tend to approach what we call the Turing limit, the point at which the parameterization becomes so extensive that it is comparable in power to a general-purpose language. "Turing" refers to the fact that a fully generalized minilanguage permits construction of any desired program, and thus is akin to a Turing machine. At best, a parameterization scheme that has reached the Turing limit will be as complex as a general-purpose language, and it could easily be far more complex.

At that point, the reuse potential of a component effectively becomes nil, since it would probably be easier and less costly to redevelop the component in the original language.

As Figure 7 shows, the best payoff in parameterization comes from finding combinations of parameters that cover common application variabilities while simultaneously requiring the least possible specification efforts from consumers. Just as many manufacturing disciplines have developed sets of complementary parts that can be adjusted and combined to produce wide ranges of useful products, good software-parameterization schemes should provide versatile tool kits of options by which the needs of later consumers can be succinctly specified.

Options that are very likely to be needed should be made as simple to specify as possible, while increasingly less likely variations should be made to require proportionally larger specification efforts. At the lower end of this domain-specificity continuum is the language itself, in which variations with very low priorities can be coded directly.



```
For i in 1 .. 3200 loop;
 If file(i) = ' ' then
  If file(i+1) = 'w' then
   If file(i+2) = 'r' then
    . . .
    file(i+1) = 'r';
    file(i+2) 'l';
    . . .
   End If;
  End If;
 End If;
End Loop;
```

```
DataStructure = File
    Format = Sequential,
    CharType = Ascii,
    RAMBuffered = Yes;
Operation = FineReplace,
    BeginDelim = ' ',
    EndDelim  = ' ',
    Find = "wrong",
    Replace = "right"
```

FIGURE 7. LIMITS OF PARAMETERIZATION IN REUSE.

WHAT'S NEXT?

We need consistent, broad-spectrum methodologies that integrate reuse analysis and development methods across the development life cycle.

In fact, life-cycle integration is likely to be one of the key factors that makes reuse truly effective. After all, if the first thing developers see in their programming environments are compilers, the first thing they will be tempted to do is to write code. If the first thing they see are lists of existing parts that may match their needs, the first thing they will be tempted to do is reuse. Put more colloquially, if you want to lose weight you should put the healthy food at the front of the fridge and the junk food at the back.

Reuse-oriented automation. Figure 8 shows the broad structure of a tool that would use instance spaces as the basis for presenting reusable components to reuse consumers[9] and that would implement separation concepts similar to those of Vic Basili and Dieter Rombach's reuse factory.[10]

The idea is to deemphasize how reusable parts are retrieved or created and emphasize the cost of retrieving them. Incorporating cost issues at such a fundamental level would let the system take a much

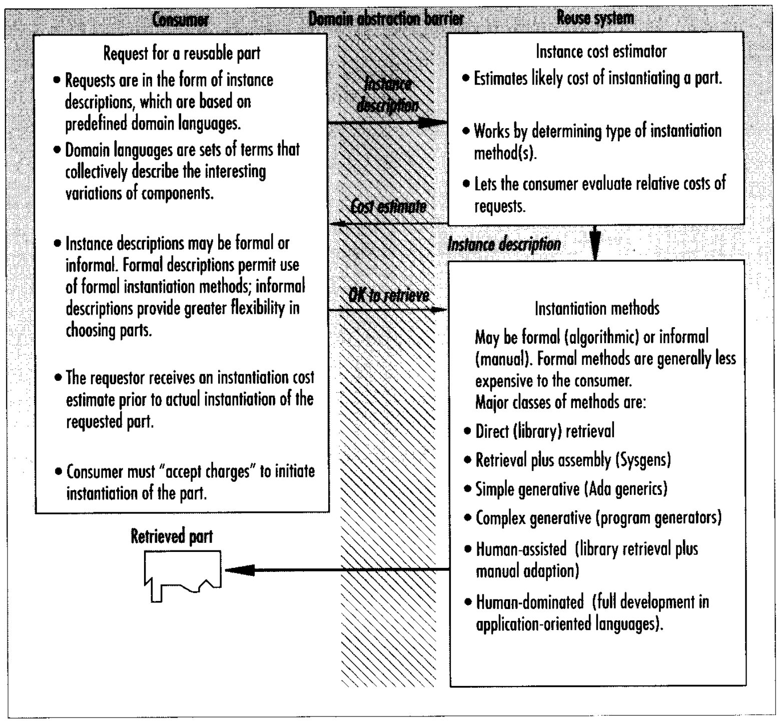

Consumer
Domain abstraction barrier
Reuse system
Request for a reusable part
• Requests are in the form of instance descriptions, which are based on predefined domain languages.
• Domain languages are sets of terms that collectively describe the interesting variations of components.
• Instance descriptions may be formal or informal. Formal descriptions permit use of formal instantiation methods; informal descriptions provide greater flexibility in choosing parts.
• The requestor receives an instantiation cost estimate prior to actual instantiation of the requested part.
• Consumer must "accept charges" to initiate instantiation of the part.
Instance description
Cost estimate
OK to retrieve
Instance cost estimator
• Estimates likely cost of instantiating a part.
• Works by determining type of instantiation method(s).
• Lets the consumer evaluate relative costs of requests.
Instance description
Instantiation methods
May be formal (algorithmic) or informal (manual). Formal methods are generally less expensive to the consumer.
Major classes of methods are:
• Direct (library) retrieval
• Retrieval plus assembly (Sysgens)
• Simple generative (Ada generics)
• Complex generative (program generators)
• Human-assisted (library retrieval plus manual adaption)
• Human-dominated (full development in application-oriented languages).
Retrieved part

FIGURE 8. REUSE ABSTRACTION: SEPARATING REQUESTS FROM METHODS.

more active role in making reuse decisions. For example, consumers who reuse early work products such as specifications or designs would receive cost quotes that would indicate the strong relative cost advantages for their selections. This type of "shop around" approach would encourage reuse consumers to select development paths that make the best possible use of existing work products.

For some time, reuse has suffered from an image problem. Anyone who has ever gone to an auto salvage yard to pick up a spare part for his old car "knows" what reuse is, and the image that it thus invokes is not altogether favorable.

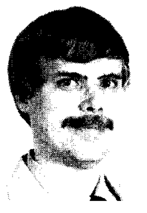
The theme we most want to convey is that reuse is not a trivial concept. As a mechanism for preserving and guiding the use of that most expensive of resources, human creativity and ingenuity, software reuse is a field that merits careful attention both from managers interested in the bottom line and from researchers interested in better understanding that most curious of symbiotic relationships, the one that exists between humans and computers. ♦

References

1. M.D. Lubars, "Wide-Spectrum Support for Software Reusability," in *Software Reuse: Emerging Technology*, Will Tracz, ed., CS Press, Los Alamitos, Calif., 1988, pp. 275-281.
2. V.R. Basili and H.D. Rombach, "Towards a Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment," Tech. Report CS-TR-2158, Dept. of Computer Science, Univ. of Maryland, College Park, Md., Dec. 1988.
3. T. Biggerstaff and C. Richter, "Reusability Framework, Assessment, and Directions," *IEEE Software*, March 1987, pp. 41-49.
4. F.J. Polster, "Reuse of Software through Generation of Partial Systems," *IEEE Trans. Software Eng.*, March 1986, pp. 402-416.
5. L. Latour and E. Johnson, "Seer: A Graphical Retrieval System for Reusable Ada Software Modules," *Third Int'l Conf. Ada Applications and Environments*, IEEE, Piscataway, N.J., May 1988, pp. 105-113.
6. P.G. Bassett, "Frame-Based Software Engineering," *IEEE Software*, July 1987, pp. 9-16.
7. M.A. Simos, "The Domain-Oriented Software Life Cycle: Towards an Extended Process Model for Reusability," in *Software Reuse: Emerging Technology*, Will Tracz, ed., CS Press, Los Alamitos, Calif., 1988, pp. 354-363.
8. M. Lenz, H.A. Schmid, and P.F. Wolf, "Software Reuse through Building Blocks," *IEEE Software*, July 1989, pp. 34-42.
9. T. Bollinger and B.H. Barnes, "Reuse Rules: An Adaptive Approach to Reusing Ada Software," *Proc. Artificial Intelligence and Ada Conf.*, George Mason Univ., Fairfax, Va., 1988, pp. 14-1–14-8.
10. V.R. Basili and H.D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Trans. Software Eng.*, June 1988, pp. 758-773.

Bruce H. Barnes is deputy director of the division of computer and computation research at the National Science Foundation. His main research interest is software research.

Barnes received a BS, MS, and PhD in mathematics from Michigan State University. He is a member of the American Mathematics Society, IEEE Computer Society, and ACM.



Terry B. Bollinger is a senior member of the technical staff at the Contel Technology Center. He works in the software-engineering laboratory on applied software research topics ranging from reuse and software maintenance to cost-oriented modeling of software processes.

Bollinger received a BS and MS in computer science from the University of Missouri at Rolla. He is a member of the IEEE.

Address questions about this article to Barnes at Division of Computer and Computation Research, NSF, 1800 G St. NW, Washington, D.C., 20550; Internet bbarnes@note.nsf.gov, or Bollinger at Contel Technology Center, 15000 Conference Center Dr., Chantilly, VA 22021-3808; Internet terry@ctc.contel.com.