Balancing Software Composition and Inheritance to Improve Reusability, Cost, and Error Rate

William R. Bitman

Software coupling—the interdependence among software components—directly affects error rate and maintenance costs. Although reuse can reduce errors and cut initial development expenses, many reuse techniques such as inheritance (in which components derive behavior from ancestors) increase coupling. In contrast, composition—the combination of independent (i.e., noncoupled) components into larger units—promotes reuse without coupling. This article formulates models that show why coupling should be avoided and how compositional techniques result in higher reusability. However, the use of composition to the exclusion of inheritance is not always advisable because the former has associated costs whereas the latter has important benefits. Therefore, a balance that combines the strengths of both is preferable. For graphical user interface systems at APL, window classes have proven to be good candidates for inheritance. Classes for business rules, however, are designed as composable components that maximize their reusability without compromising independence.
(Keywords: Inheritance, Software complexity, Software composition, Software coupling, Software reuse.)

INTRODUCTION

Software reuse, i.e., the use of existing software components to construct new software, is one of the most promising approaches for boosting software development productivity, both within a system and across systems.[1] In most software systems the degree of functional overlap and code redundancy is such that the application of reuse techniques creates significant improvements in the software development process and in the products produced.[2–5] For example,

- A NASA software reuse team found that 30% of existing components were relevant to new projects, and 80% of those components did not need modification. Their reuse program achieved a concomitant decrease in development cost.[6]
- Toshiba's practice of code reuse lowered error rate from 20 to 3 defects per 1000 lines of code over an 8-year period.[7]
- Hughes Aircraft achieved a 37% development cost savings through reuse.[8]
- Hewlett-Packard's organized effort to define and build reusable software components eliminated the need to write 10% of code.[9]
- A French Navy and Army software reuse program resulted in a 34% reduction of new code development, with a 33% drop in error rate.[10]

- Raytheon achieved a 50% productivity increase for software projects selected for reuse.[2]

Unfortunately, not all reuse programs have been successful.[11,12] Reuse techniques cannot save a poorly managed project. In fact, reuse itself is a process with its own organizational and managerial requirements (e.g., time, effort, knowledge).[13–15] To be reused, the software must fit the requirements specified for the application to be developed, be designed in a reusable form, and be locatable by developers.[16]

A factor that works against reuse is that some environments are varied and volatile. Reusable software designed and developed under such conditions often never has an opportunity to be reused. Frustration ensues when an organization realizes that its earlier attempts at reuse were in vain. Those that have been successful at reuse may have selected a narrowly focused domain and may have done so during periods of tranquillity. In fact, the ability to expend the extra effort needed to make software reusable may be feasible *only* during relatively tranquil times.

Reuse protocols such as object linking and embedding (OLE) and common object request broker architecture (CORBA) have helped circumvent these difficulties for information systems. These standards have made reusable text editors, spell checkers, word processors, and spread sheets available for many platforms and environments. For some organizations, this is the only type of reuse that is cost-effective; the additional expense of more involved reuse efforts offers no payoff. However, organizations that invest heavily in custom-built applications will benefit from the reuse of their domain-specific software. Therefore, it is worth investigating the nature of software reuse.

This article focuses on a single aspect of reuse: how the form of code at its most fundamental structural level affects software reusability. Although we will deal with software as a product here, software improvement may be more readily achieved through studying the process of software development rather than its product. Furthermore, studying the elemental unit of the product does not create a complete picture of the finished product (e.g., an executable application) because of the many complexities and dynamics that the delivered product exhibits over and above those of its elemental units. Nonetheless, an understanding of the elemental unit of the product offers insights into the nature of the application as a whole and the types of process improvements most likely to be effective.

The purpose of this article is twofold: (1) to explain why certain forms of reuse have detrimental side effects and (2) to present an approach that enables designers to realize the benefits of reuse while minimizing disadvantages. First, quantitative studies in the software engineering literature are reviewed and models of factors that affect cost and error rate are formulated from them. Next, the mechanism and advantages of techniques that lower cost and error rate are explained. Finally, guidelines for achieving a balance of techniques that have conflicting attributes are given, and an example of a highly reusable component designed and developed at APL for a specific application is described.

THE EFFECT OF COUPLING, COMPLEXITY, AND COHESION ON ERROR RATE AND COST

Over the past 20 years, software engineers have discovered a relationship between software cost and error rate. Figure 1 is a model formulated as a composite of results from dozens of published studies. Coupling (references from one software component to another) is a central item in the model. Although most popular programming courses and books discuss encapsulation and decoupling, this article explains the mechanism behind these concepts by integrating the results of numerous published studies into a single model. An understanding of this model can help guide and motivate more developers to find ways to integrate engineering techniques into their everyday work. In the following paragraphs, the factors (boxes in Fig. 1) in the model are defined and their relationships (lines in Fig. 1) are explained.

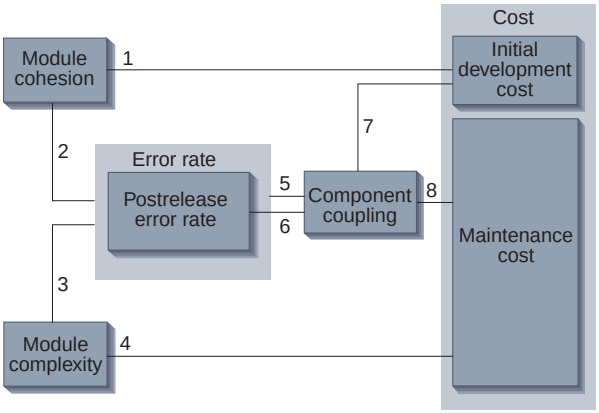

Figure 1. Model of software code factors that affect error rate and cost. Numbered lines represent relationships found in published studies. The lines, however, do not have arrowheads because the research findings do not imply causality. Both error rate and cost are indirectly related to module cohesion and directly related to module complexity and component coupling. Postrelease errors are more costly to fix than other errors and are therefore represented by a box that is more than half the size of the total error rate box. Since maintenance is costlier than initial development, it too is represented by a larger box. Component coupling is directly related to both postrelease errors and maintenance costs.

Factors

The associations among the factors in Fig. 1 have been found to be statistically significant by various research studies, as will be indicated in the next section on relationships. The term "module" in our model means the smallest nameable unit of code (e.g., a function), not an executable application as it is sometimes defined. "Component" refers to a structure that is a step higher in scope than module but still below the application level. The metrics associated with software at the module and component levels are also fundamental and do not include larger measures such as function points (a measure of the size of functionality units from the application customer perspective) or application development cost-estimating models. Instead, metrics at the levels discussed here can serve as inputs to these systemic measures.

Module Cohesion

A module is an elemental, callable unit of code. Functions and class methods are modules. The concepts of "class" and "object" are used throughout the article and are defined here.

A class consists of data members and methods that represent state and behavior, respectively.[17,18] An object is an instance of a class. A software object is the abstraction of something that can be expressed as a noun: a person, place, or thing. It can be a real-world object modeled in software, such as "employee," or it can be a concept, such as a doubly linked list. A "method" is a function found as part of a class. It represents a behavioral aspect of the object being abstracted. A string search method is a function that is a member of class "text," for example. For this discussion, both traditional classes and Ada packages (a language construct in which modules are grouped into a single named component) are referred to as classes,[19] even though Ada packages can be utilized for non–object-oriented purposes.

"Cohesion," also known as strength, refers to how well all parts of a module contribute to the performance of a single well-defined task. For example, a curve-fitting module should not directly output a physical representation of its result because different output devices describe line segments and curves differently. Each output function should be a separate method. Similarly, in information systems, business rule logic should be distinct from display algorithms. Cohesion is difficult to define mathematically, and most studies quantify it by employing ranking techniques performed by software professionals.[20]

Module Complexity

Module complexity can be calculated by using many approaches, including number and types of operators and the extent of branching and looping. Although no approach has a rigorous derivation,[21] many studies have adopted definitions based on branching and looping. A notable example is McCabe's[22] cyclomatic complexity metric, which has been found to have significant relationships to error and cost, as will be shown.

Component Coupling

A component is a discrete unit of code that can be transferred in its object form (e.g., `.obj file`) among environments. Functions, libraries, and classes are components. Coupling is created by references and calls from one component to another. This includes references to global variables and data members of another class as well as calls to functions or methods of another class. The more calls and references, the higher the coupling. However, calls among methods within a class do not increase coupling because a class is a component (see the section entitled Calling Techniques). The opposite of coupling is independence.

An important attribute of an independent component is "encapsulation." An encapsulated software component can be used correctly and effectively solely through its external definition and declaration, without the need to understand its internal characteristics or facts about other components. A component's extent of encapsulation is one indication of its degree of independence.

Different terms for coupling are used in the research literature on software engineering. "Fan-out" indicates that a functional component fans out to numerous calls to other components, thereby dividing the labor. "Design complexity" denotes the structure created by calls among numerous components. "Interface complexity" can refer to the number of calling parameters in a module declaration, but the same term can also be used to mean the number of calls among components. Therefore, some studies that deal with fan-out, complexity, or both are, in fact, measuring coupling.

Error Rate

Error rate is the average number of defects per (usually) 1000 lines of code. Whenever studies do not specify pre- or postrelease errors, total error rate is assumed. This, again, is represented by the large error rate box in Fig. 1, which encompasses the postrelease error rate.

Many studies focus on errors that are detected only after release of software. The longer it takes to identify an error, the costlier it is to fix.[23,24] Therefore, initiatives to establish methods to avoid postrelease errors (the larger portion of the error rate factor in Fig. 1) deserve special attention.

Costs

Initial development cost is the cost to bring an application to its initial release. Total cost is the sum of initial development and maintenance expenses.

Maintenance cost, which is based on the ANSI/IEEE 729 standard definition of maintenance,[25,26] includes postrelease expenses incurred to fix defects, improve performance, add enhancements, and adapt code to match changes in the environment. Within conventional environments, maintenance of existing systems is the costliest aspect of the development process, shifting most resources away from the creation of new applications.[27–30] Thus, in Fig. 1, the maintenance cost box is larger than the initial development cost box.

Relationships

The model in Fig. 1 is based solely on actual study results, not on inferences. For example, according to the standard definition of maintenance cited previously, postrelease error correction is included in maintenance, thereby creating an implicit relationship between the two factors. The model, however, does not link them because the relationship is known by definition rather than by experimental studies. In contrast, line 6 (between component coupling and postrelease error rate) and line 8 (between component coupling and maintenance cost) are included in the figure because different studies measured each of these and found the relationships to be statistically significant.

In summary, the relationships represented in our model (Fig. 1) are as follows:

1. Initial development cost is indirectly related to module cohesion.[20]
2. Error rate is indirectly related to module cohesion.[31]
3. Error rate is directly related to module complexity.[32–34]
4. Maintenance cost is directly related to module complexity.[30]
5. Error rate is directly related to component coupling.[31,35–39]
6. Postrelease error rate is directly related to component coupling.[24]
7. Initial development cost is directly related to component coupling.[31]
8. Maintenance cost is directly related to component coupling.[26,30]

Some early studies implicated module size measured in lines of code as being directly related to error rate and cost, whereas others found no such relationships. Subsequent studies were more carefully controlled for module complexity and cohesion. These studies indicated that size, rather than being related to errors and cost,[30] was merely a covariant of coupling and/or complexity in some environments due to the nature of the domain or the software coding approach.

Interpretation

The lack of arrowheads in Fig. 1 indicates that the research findings do not imply causality. For example, even though maintenance cost is directly related to component coupling, a decrease in component coupling does not mean a decrease in maintenance cost. Instead, coupling and complexity may be symptoms or artifacts of other factors, such as project size, project complexity, design technique, or programmer experience. However, the more recent studies cited have controlled as many of these factors as possible. Therefore, the published findings, for purposes of this article, will be interpreted as implying these relationships. Thus, the model states that if we can increase cohesion, decrease complexity, and/or decrease coupling, we can decrease cost and errors.

Managing the simultaneous optimization of coupling, complexity, and cohesion can be too difficult to put into practice. Fortunately, intrinsic relationships exist among these factors that simplify the task and enable formulation of a manageable approach. Increasing cohesion can be accomplished by removing code that is extraneous to the activity at hand and instead creating a separate module for it. This very process also reduces complexity, enabling designers to concentrate on optimizing either cohesion or complexity, rather than attempting to control both simultaneously. Cohesion is largely subjective,[20] whereas complexity is easy to measure. Therefore, a practical and effective approach could be to use intuitive cohesion-strengthening techniques as a guide for reducing complexity. With such an approach, cohesion becomes a working criterion for design decisions, whereas complexity is the attribute that is measured and monitored as an indicator for both complexity and cohesion.[40] This narrows the overall objective to the reduction of complexity and coupling.

Much effort was spent in the 1970s and 1980s to find optimal ways to reduce complexity by decomposing software through partitioning.[41–43] A certain level of success has been achieved with this approach. For example, a cost-reduction program for a Navy attack aircraft software system achieved a 10% reduction in

schedule and cost simply by partitioning software components.[26] Also, complexity has been used as a feedback mechanism for partitioning.[24,44] In addition, by reengineering legacy systems via partitioning, a measurable decrease in complexity has been achieved.[45]

The problem with partitioning is that, within traditional (i.e., non–object-oriented) environments, as each module becomes less complex and more cohesive, coupling increases. For example, a large module can be divided into two modules, thus distributing the complexity. However, each instance of partitioning necessitates at least one call from one module to another, which increases coupling. On projects that adopted partitioning, modules were indeed simplified in the early developmental phases as expected, but these techniques only deferred complexity to later phases of the project and, more importantly, caused coupling to increase. Consequently, as deadlines approached, projects were hopelessly swamped in more complexity than ever. Excess coupling added to the difficulty of untangling that complexity.

An opposite approach is to concentrate on decreasing coupling. Taken to an extreme, an entire application can be contained in a single module, thereby eliminating coupling completely. The resulting complexity and lack of cohesion are unacceptable, however. The nature of object classes makes them an especially effective vehicle for increasing cohesion while reducing complexity, since algorithms can be decomposed into their elemental activities and separated into individual methods, although the result is still a single component. Thus, a class is a powerful engineering structural unit. Ultimately, however, since components must communicate and interact, we need more comprehensive techniques to reduce coupling without increasing complexity.

COUPLING REDUCTION

Table 1 compares various software reuse techniques, which are divided into three groups on the basis of their reuse mechanism, i.e., compositional techniques, techniques that achieve reuse through calls from one module to another, and inheritance techniques. The techniques are listed in descending order of engineering rank. Compositional techniques rank high in engineering value. Techniques that enable calls without coupling rank higher than those that do not. Classes are notable in that calls within classes do not introduce coupling. Inheritance, which is not necessarily a trait of classes,[17] is treated separately. Inheritance ranks lowest in ability to encapsulate. As seen in the table, a low coupling rank indicates high independence. Each technique will be described in the following paragraphs, and the overall reusability of languages incorporating them will be compared. First, the rationale for the ranking is explained.

The ranking of these techniques is derived from the values of three major characteristics—composition, decoupling, encapsulation—plus other considerations, which are given in column [5] of Table 1. Column [1] indicates whether the techniques represent a compositional process. The top two techniques (rows [A] and [B]) are compositional. In column [2], the techniques are evaluated in terms of decoupling. For this evaluation, calls to built-in modules, classes, or libraries for a language are not considered coupling, even though they limit software to that language. (This is a commonly accepted limitation, although object files of C functions and libraries are not restricted because they are callable from most other languages.) On the basis of this definition of coupling, the top five techniques (rows [A–E]) are decoupled.

The ability of a technique to encapsulate is indicated in column [3], which reveals a major shortcoming of inheritance, i.e., it is the only technique that prevents encapsulation by its very nature, as will be explained.[17] Within groups of techniques having equal values for the three major characteristics (composition, decoupling, encapsulation), ranking is based on factors presented in column [5] in terms of scope of reuse and difficulty of reuse. These additional criteria are used to produce the final ranking order.

Composition

The top two reuse techniques in Table 1 are compositional. The mechanism of composition via “generic formal parameters” (row [A]) is diagrammed in Fig. 2. The component of interest (in the center of the figure) achieves independence by not calling any physical modules or referencing any physical data items. Instead, surrogate data types and modules are defined as generic formal parameters. To utilize the component in an application, a developer substitutes physical methods and data types for the generic formal parameters. This is referred to as an instantiation, which consists of references, shown in Fig. 2 by arrows, to the independent components. Instantiation does not change the independence of any of the components.

Note that “generic” is used in two different ways throughout the article. The term “generic formal parameter” is a technical term that is part of the definition of the Ada language. When used by itself, the word takes on its common meaning, namely, relating to a whole group. Both terms imply independence of software components.

Generic formal parameters have many benefits. First, a developer can mix and match components, which makes them “composable.” Second, source code

Table 1. Comparison of reuse techniques.

Reuse technique (type)	[1] Composition	[2] Fully decoupled within language	[3] Encapsulated	[4] Coupling rank	[5] Rationale for rank order	[6] Sample language[a]
[A] Generic formal parameters (composition)	Yes	Yes	Yes	1	Totally decoupled. Large scope of reuse within the language. Components must be available as substitutes for generic formal parameters, but can be freely selected from numerous components.	Ada
[B] Custom user objects (composition)	Yes	Yes	Yes	2	Compositionally combines classes into larger, more functional classes. Moderate scope of reuse.	PowerBuilder
[C] Classes (calling)	No	Yes	Yes	3	Extremely decoupled, but reuse is limited to calls and references made within the component.	C++
[D] Standard classes (calling)	No	Yes	Yes	4	Reuse without coupling within language/ compiler/operating system domain. Resulting modules can be highly callable across languages via standard interface protocols.	C
[E] Built-in functions (calling)	No	Yes	Yes	5	Reuse without coupling within language/ compiler/operating system domain.	Fortran
[F] Reference to global (calling)	No	No	Yes	6	To use a component, declare and initialize the global variable, edit the code to reference a global already in the application, or parameterize the component.	All
[G] Call or reference to an external component (calling)	No	No	Yes	7	Highly coupled. To use, include all references.	All
[H] Inheritance (inheritance)	No	No	No	8	Most tightly coupled. Cannot correctly use a descendant object without knowing everything about all its ancestors.	C++ PowerBuilder Ada95

[a]Language that exemplifies use of this technique.

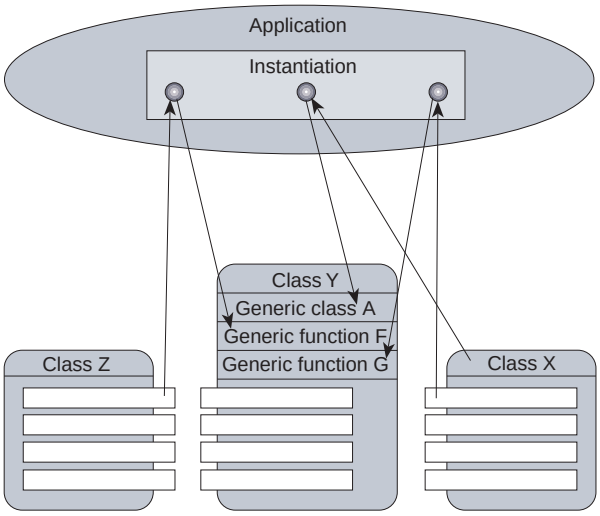

Figure 2. Mechanism of composition via generic formal parameters. In Ada, these parameters eliminate the need to call external modules. In the application, actual modules and types are defined in the instantiation. Developers compose functional components by combining simpler components. The arrows indicate that the instantiation statement creates a new object by combining independent classes.

never needs to be duplicated. For example, code for date methods resides only in a date class. Similarly, all code for text manipulation is found in a text class. Any component that needs date or text algorithms avoids coding them by defining them as generic formal parameters. Third, when changes are needed to a component, they are made in one place only. All instantiations will register the changes. A fourth benefit is that when changes are made to the operational requirements of an application, existing components are not modified. Instead, the instantiations are modified and new ones are added as needed. These benefits explain why generic formal parameters boost reuse, which was one of the original visions and motivations for the definition of Ada.[46] The Ada generic formal parameter is a model for defining highly reusable independent components through composition.

Components, although independent, must be able to be combined in order to be usable and useful. This requires planning and may limit the freedom to mix and match components in real-world situations. For example, suppose a component declares generic formal parameters for various operations on dates, including the ability to find the next Monday after a given date. A date class that does not contain this method, even if preferred for its other features, will not be usable for the instantiation. Instead, the developer must obtain the source code for the preferred date component and add the needed method; settle for some other date component that has all the needed methods but may lack certain desired qualities such as fast execution; or code a new component from scratch that has all the desired qualities. Another difficulty is that compilers and run-time environments for Ada typically produce applications with unacceptably long response times when software uses advanced features such as generic formal parameters. This problem has been remedied in recent years with the advent of more efficient Ada systems. However, given the potential for reuse that generic formal parameters provide, these practical factors help explain why the much-anticipated reuse explosion from this technology never occurred.[47]

Theoretically, however, composition via generic formal parameters provides the greatest potential for reuse. Therefore, developers should seek features that emulate this paradigm in whatever language or environment used. Custom user objects (row [B] of Table 1) in the PowerBuilder language approach the compositional nature of generic formal parameters. They are formed from standard classes and can themselves be combined into larger functional classes. But no mechanism exists to substitute classes the way generic formal parameters allow, which results in the need to duplicate some code to achieve independence. For example, if two classes must perform a string search, calling a method from a text class will create coupling. In Ada, this is avoided by declaring a generic formal parameter string search. In languages that have no such mechanism, one can avoid coupling only by including the method physically in each class that needs it. In its favor, however, PowerBuilder also provides many attributes and methods that are generic across objects, thereby increasing the ability to achieve independent classes that can be placed freely into various applications as needed.

Calling Techniques

Table 1 includes five techniques (rows [C–G]) that call modules or reference variables. Figure 3 shows examples of calls that create coupling and calls that do not. Coupled techniques are denoted by red arrows, whereas the black arrows are calls that do not increase coupling. The only black arrows in the figure are those for intraclass calls and references, because calls among modules within a class and references to data members within a class do not introduce coupling. This is a powerful way in which classes encourage good engineering. In Table 1, the first three of these techniques (rows [C–E]) are independent, whereas the last two ([F] and [G]) are coupled, as shown in column [2].

Function libraries also enable noncoupled calls because the entire library must be transferred to an application as a unit. Libraries of classes compound this benefit even further because calls within classes as well as calls among classes within the library are not coupling. However, calls among components within a

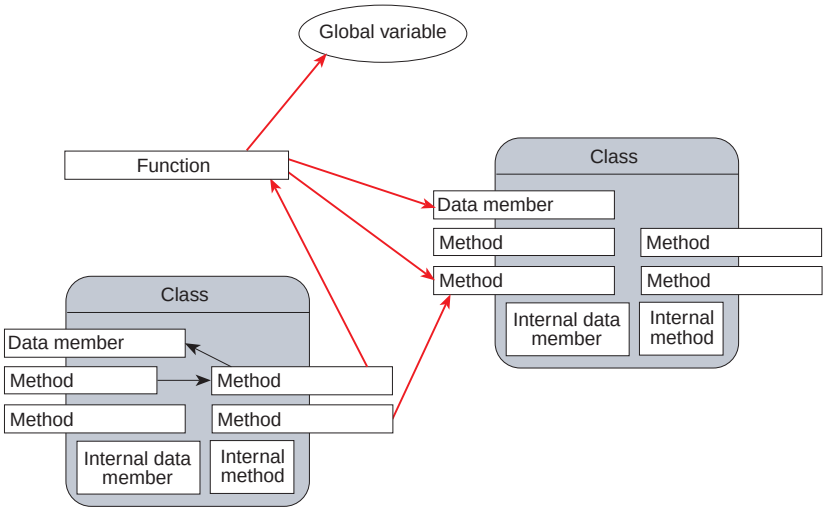

Figure 3. Examples of types of calls and references. Red arrows indicate calls and references that span components, thereby increasing coupling. Black arrows are calls and references within a component (i.e., class) and show reuse without coupling.

library limit the freedom to mix and match separate components. Therefore, only classes, not libraries, represent this technique in the model. This does not imply that libraries of classes cannot be used, however. Classes, whether stand-alone or in a library, that contain only internal calls and references are equally independent. The exclusion of libraries for this discussion refers only to the idea of considering a library as one large component, in which calls among different member components are not considered coupling.

It is important to differentiate between two characteristics of classes. The first is the ability to make calls internally so that coupling is not increased, as stated previously. The second is that most languages that support classes also support inheritance. Many believe that a language must offer inheritance to be object oriented. In fact, this is not true.[17] Classes and inheritance are not synonymous, and in this article classes are used to represent the ability to increase reuse and decrease complexity without increasing coupling. Inheritance is treated as a separate technique.

Inheritance: Loss of Encapsulation

With the increased growth of object-oriented programming, inheritance has become a popular reuse technique. The immediate benefits are easy to demonstrate; e.g., behavior and attributes common to numerous classes can be placed in an ancestor and never need to be coded again.

However, inheritance has drawbacks that appear only during later phases of development and maintenance. The major difficulty is that, as shown in Table 1, inheritance nullifies encapsulation (column [3]). An object that inherits from an ancestor requires developers to understand the entire ancestor lineage for that object, a fact that goes against modern engineering principles.[17] Second, descendants often inherit behavior that is incorrect for them owing to requirements that are unique for different descendant objects. In such cases, the code in the ancestor must be suppressed and subsequently becomes useless baggage. In practice, this occurs often because the mechanism of biological taxonomic classification that serves as the paradigm for inheritance is not well suited for software.[17] Third, additions and changes to requirements make a previous inheritance architecture obsolete. It is quite labor-intensive to redesign ancestors to fit new requirements and forces developers to re-create descendants as well, thus adding to the effort. However, the seemingly expedient alternative, i.e., creating workaround code, results in complex, splintered layers of code whose control flow is hard to follow. This makes inherited software difficult to learn, use, and maintain. Any quick gains made from inheritance are lost in later developmental phases and during maintenance.

Reusability

Figure 4 shows the effects of applying decoupled reuse techniques (rows [A–E] of Table 1) on complexity as exemplified by the sample programming language selected in column [6] of Table 1. Each language, because of the nature of the decoupled reuse techniques available, limits developers as to how much they can reduce average complexity.

The rationale for using average McCabe cyclomatic complexity is as follows. All systems contain modules that have low cyclomatic complexity because they have simple control flows. However, when the complexities of all modules of a system are averaged, poorly designed systems have high average complexity values. Improving design and coding techniques reduces the complexity of previously complex modules, thereby lowering the average complexity.

Starting from the most basic language, Fig. 4 charts a progression of engineering techniques. As techniques progress in engineering value, the optimization point moves closer to the region of low coupling and low complexity. (The optimization point is the language's line intersection with the optimization line and

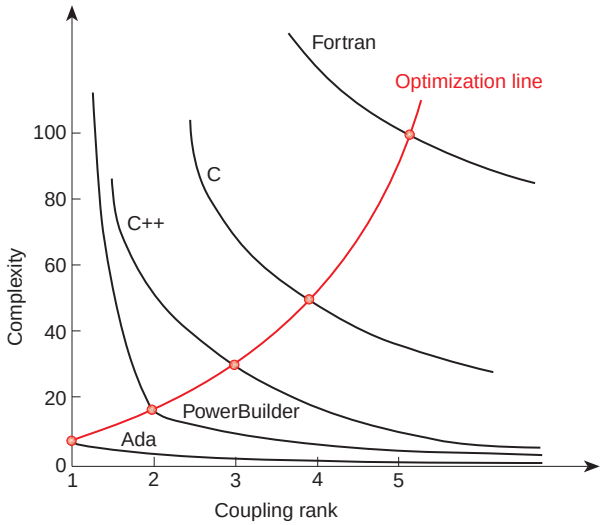

Figure 4. Sample language comparison based on the language's ability to enable developers to use decomposition techniques while keeping coupling low without increasing complexity. Coupling rank is taken from Table 1, column [4]. Circles along the optimization line represent optimization points. Complexity values (from quantitative studies[45] and unpublished measurements taken by the author) are based on the average McCabe cyclomatic complexity that is achievable from using the engineering features of the language. To the left of the optimization points, a decrease in coupling produced by combining modules (aggregation) causes an increase in complexity. To the right of those points, a decrease in complexity via partitioning techniques produces an increase in coupling.

represents the point at which a decrease in complexity does not cause an increase in coupling and vice versa. The optimization line is created by fitting the optimization points.) In Fortran, the lack of classes, packages, and standard libraries creates an increase in complexity as decomposition occurs. Complexity can be lowered, but only at the expense of coupling, and vice versa. The C language adds standard libraries. This creates a bend in the middle of its line at which point coupling and complexity do not negatively impact each other (see the circled optimization points in the figure). C++ adds classes so that internal calls create a line that is more conducive to coupling and complexity reduction. PowerBuilder provides standard classes, calls to which do not increase coupling, and custom user objects for compositional design. Ada has generic formal parameters, the paradigm of independence and composition.

Although there is no absolute preferred high limit for module complexity value, empirical studies,[34] including those for the Aegis Naval Weapon System,[48] have concluded that error rate decreases significantly when the average complexity does not exceed a value of 10. Figure 4 shows that Ada generic formal parameters, with an expected best average complexity value of 7, readily help developers achieve this limit. Custom user objects (e.g., PowerBuilder), with a value of 14, enable software to come close to this ideal.

Table 2 calculates a reusability index (column [4]) based on the lowest average complexity attainable when capitalizing on the decoupling techniques available in each language. The index is derived as complexity (column [3]), which is taken from the optimization point on Fig. 4 for each language, divided by the coupling rank in column [2], which is taken from column [4] of Table 1. Independent, composable software components, such as classes with generic formal parameters and custom user objects, impart a high reusability (low index value in column [4], Table 2). The reusability index quantifies what is shown graphically in Fig. 4, namely that compositional techniques enable designers to achieve lower values of complexity and coupling. Software developed in languages that provide less rigorous reuse techniques, on the other hand, can attain only limited reusability.

Table 2. Reusability index.

Technique	[1] Sample language	[2] Coupling rank	[3] Average complexity[a]	[4] Reusability index[b]
[A] Generic formal parameters	Ada	1	7	7
[B] Custom user objects	PowerBuilder	2	14	7
[C] Classes	C++	3	30	10
[D] Standard libraries	C	4	50	12.5
[E] Built-in functions	Fortran	5	100	20

[a]Achievable when engineering techniques are fully utilized.
[b]Coupling/complexity: [4] = [3]/[2].

BALANCING COMPOSITION AND INHERITANCE

Composition (generic formal parameters) and inheritance each have advantages and drawbacks (Table 3). The choice of technique depends on such factors as time allotted for development (row [D]), expertise available (row [E]), the expected life span of the application, and how actively it will be maintained (row [F]). Generic formal parameters and user objects, being compositional reuse techniques, share the following advantages: they result in independent, composable components; the resulting components are encapsulated; and, based on the model, they have the potential to lower maintenance costs. Whether maintenance costs actually will be lower depends on the volatility of the environment and the nature of the changes. On the whole, however, there is a greater probability that a system can be adapted in response to changing requirements without rewriting existing parts. Inheritance does not exhibit these advantages. However, the disadvantages of compositional techniques are that they require greater expertise and take longer to develop. Although custom user objects may not eliminate duplication of code, generic formal parameters do. Inheritance also eliminates duplication of code, shortens development time, and demands less expertise; it does not, however, encapsulate software, and higher maintenance costs can be expected.

How should designers decide between composition and inheritance? No code duplication requirement (Table 3, row [C]), shorter development time (row [D]), and lower expertise requirement (row [E]) make inheritance an efficient approach for small applications. On the other hand, large projects whose software is expected to have an extended shelf life will benefit from compositional techniques owing to lower error rate and maintenance costs (row [F]). However, even for large applications, the benefits of inheritance should not be ignored. Creating an overall inheritance architecture within which compositional components are used can provide a real-world compromise that enables designers to capitalize on the advantages of both techniques.[17]

APL has adopted this balanced approach for some projects. A hierarchy of ancestor windows forms a framework in which basic attributes are set at the correct inheritance level. However, algorithms are not hard-coded in ancestors. Instead, most software is produced as independent composable classes. These components can be placed on any of the windows in the system. The task of switching from one inheritance hierarchy to another, if needed because of changing requirements, is somewhat eased since the independent components can be placed within any inheritance framework.

Table 3. Comparison of major design techniques.

Reuse technique trait	Generic formal parameters	User objects	Inheritance
[A] Composable	Yes	Yes	No
[B] Encapsulated	Yes	Yes	No
[C] Code duplication requirement	No	Yes	No
[D] Shorter development time	No	No	Yes
[E] Lower expertise requirement	No	No	Yes
[F] Lower maintenance cost	Yes	Yes	No

INDEPENDENT, REUSABLE COMPONENTS AT APL

APL's Computer Applications Group in the Business and Information Services Department modified legacy applications to a client server architecture in an effort to move software off the mainframe a few years ago. This was viewed as an opportunity to select a language that supports compositional techniques. For graphical user interface (GUI) applications, the PowerBuilder development environment provides numerous standard GUI classes, includes the ability to compose classes into larger functional components via custom user objects, and provides many attributes and methods that are generic across classes to maximize independence.

Originally, the PowerBuilder product supported application development only for Microsoft Windows environments, although Macintosh and Solaris versions are now available. However, users with non-Windows platforms at APL are supported by the original applications, which are accessed through a server that provides emulation. Classes developed with custom user objects can be used in the C++ environment via the Optima++ product released by Sybase, extending even further their reusability.

The specific class developed at APL using composition, although highly specialized and technical in nature, provides a concrete example of the real-world application of the various design principles explained in this article.

Compositional Design Example: Reusable Class for the Automatic Saving of History Rows

One reusable component developed at APL is a class that automatically

saves history rows when data are modified or rows are deleted. This class functions with any database management system. During the following discussion, references to SQL (the commonly used relational database language) and procedural language statements are included to illustrate how the language features are applied to boost composition and independence.

A history row is a row in an SQL table as it existed in the past. In a system that saves history rows, when a `commit` (the statement that saves changes to database tables) is issued, a modified row is inserted as a new row, and the original row is left intact. Similarly, a row that a user deletes is not physically deleted from the system, but instead is marked as deleted. Ordinary systems keep only the current state of data, whereas a system that saves history rows is two-dimensional in that it stores data over time.

Many benefits are realized from keeping history rows. They provide a running record of "who did what when." This lowers the risk of decentralizing data maintenance because inadvertent data modifications and deletions can be corrected by restoring an earlier row. Also, reports can be generated as they would have appeared at previous points in time, and an entire database or portion of it can be restored to a previous state.

Few systems today save history rows because of increased disk space requirements, algorithm complexity, lengthened response time, etc. The component designed at APL is a reusable generic class that alleviates the algorithm complexity problem since it can readily be integrated into many systems without any additional developmental effort. It also reduces the additional disk storage space needed by eliminating duplication of stored data and minimizes response time for critical repetitive operations that users perform in real time. The solution presented here can be used not only as part of the design of new databases but can also be applied to existing tables by appending history columns. The following material is simplified to focus on the essential concepts. The actual implementation has many other features.

Throughout this discussion, history columns for the row creation time stamp and for the identification (`id`) of the operator who created the row are named `last_update_date` and `last_update_id`, respectively. In addition, `db_operation_ind` is an indicator column that allows the rows to be marked as deleted by setting its value to the letter '`D`'. The column names and deletion code value are declared in the instantiation of the object by initializing instance variables, and can therefore be used within any organization's naming convention without modification.

In an ordinary system, the primary key, referred to here as the functional primary key, is defined solely by requirements (e.g., the primary key for table customer is `customer_id`), and only one row exists for each unique primary key value. In a system that stores history rows, however, there can be many physical rows for each logical row. The current row is the latest physical row. Whereas a logical row is unique by functional primary key, physical rows are unique by a compound key formed by the functional primary key plus the time stamp. A deleted row has no current row, although there are at least two physical rows: the row itself and a more recently created row that indicates the time of deletion.

One design alternative is to define two additional columns—`deleted_date` and `deleted_id`—instead of `db_operation_ind`. Then the delete operation can be recorded in the row itself, without the need for a second row. However, more disk space will be used because most rows never get deleted, yet space is set aside on each row for an extra time stamp and `id`. As the number of rows increases, increasing amounts of disk space are wasted. So although this design alternative is ostensibly simpler, it was not selected.

Three table designs are available for storing history rows: single, column-duplicated, and row-duplicated tables. Each alternative is explained in terms of its characteristics, strengths, and weaknesses in the following paragraphs. A design is selected, and the implementation of a class that provides history row saving for that design, especially in terms of its generic and compositional character, is then explained.

Single-Table Design

In a single-table design, current rows and history rows reside in one table. This alternative can be implemented with just four steps: (1) The update method of the datawindow object (i.e., a class that serves as a graphical representation of an SQL statement) is set to `Use Update` (not `Use Delete then Insert`). (2) The `itemchanged` event (the system event that is triggered when the data value changes) of the datawindow control (a class that resides on a window and has a datawindow object as an attribute) must contain the following line of code so that a modified row is inserted and the previous row is kept as a history row, rather than being overwritten: `setItemStatus(getRow(), 0, primary!, newModified!)`. (3) `setRow()` (a method that sets the row as selected) is called in the clicked event (the system event that is triggered when the left mouse button is clicked while the cursor is over the datawindow object) so that the `getRow()` in `setItem-Status()` detects the correct row as selected. With these three steps in place, when the `update()` method is called to update the database

table, modified rows are inserted, leaving the original row untouched. (4) The db_operation_ind is set to the letter 'D' in the delete() method.

The single-table design offers several benefits. For example, the implementation does not reference any columns unique to the table. Therefore, it can be written once and placed as a user object datawindow, which is a composable class. Also, no rows or columns are duplicated, thereby minimizing disk requirements. This, then, is one way of avoiding inheritance and coupling while increasing reuse. Such an increase is accomplished through designing the class as generic so it can be placed on any window, rather than in an ancestor where it must reference physical attributes and methods of a specific window class.

This design has numerous drawbacks, however. The foreign keys (columns in a data table that refer back to another table's primary key) cannot be defined because the time stamp is part of the primary key. Also, retrieval of current rows is the slowest of the three design alternatives because it requires a nested select (a select within another select statement) with an aggregate function (a function that accumulates rows) to find the latest row for all logical rows:

```
select * from table_a t1
   where (db_operation_ind <> 'D'
   and id||last_update_date =
      (select max
      (t2.id||t2.last_update_date)
      from table_a t2
         where t1.id =
         t2.id group by t2.id))
      order by t1.id asc
```

Response time to retrieve current rows can be significantly shortened by including Boolean indicator column latest_ind. The statement to find all latest rows then simplifies to

```
select * from table_a
   where latest_ind = 'Y'
      order by id asc
```

The table design that includes this column is shown in Table 4. In all database table design figures, the names of primary key columns are underlined. Columns id and data are example data columns, and all other columns are history columns. Adding column latest_ind improves performance for selecting the latest rows for each id by avoiding the use of an aggregate function in the SQL select.

Examples of an unmodified row, a modified row, and a deleted row are exhibited in Tables 4–7. Even with the benefit of latest_ind, however, response time is still not optimal because of the extra where clause, which must extract current rows from a table crowded with history rows. More importantly, once latest_ind is added to the table, the generic four-step solution stated previously no longer works because code must be written to set the latest_ind of the old row to 'Y' without creating a new row in the process. Thus, algorithm development, coding, and testing become factors for the developmental effort, making it advantageous to evaluate other design alternatives that may overcome these problems.

Column-Duplicated Table Design

The column-duplicated and row-duplicated table design alternatives have a separate history table for each table with current data, allowing faster retrieval of current rows. In the column-duplicated table design, history columns are duplicated in the current data table. But the current data table contains only current rows, and the history table contains only history rows (Table 5). Therefore, no rows are duplicated.

This design uses even less disk space than the single-table design because there is no latest_ind column. The column db_operation_ind is not needed for current rows. It can, however, be included in the current data table to make the code generic for data transfer between current and history tables for archive and restore utilities that are common to most information systems. The history row custom user object, as implemented, functions whether the current table has the extra column or not. Since retrieval of current rows

Table 4. Database table design for storing current and history rows in a single table.

id	last_update_date	data	last_update_id	db_operation_ind	latest_ind
1	10/01/95 10:00	A	00102331		
1	10/15/95 09:00	A1	00100227		Y
2	09/03/95 11:15	B	00102200		
2	10/05/95 13:00	B	00100345	D	Y
3	09/09/95 09:09	C	00100123		Y

Table 5. Database table layout for column-duplicated design for saving history rows.

id	last update date	data	last_update_id	db_operation_ind
1	10/01/95 10:00	A	00100001	
2	09/03/95 11:15	B	00100200	
2	10/05/95 13:00	B	00110345	D

Table 6. Current table layout for row-duplicated design.

id	data
1	A1
3	C

Table 7. History table layout for row-duplicated design.

id	last update date	data	last_update_id	db_operation_ind
1	10/01/95 10:00	A	00100011	
1	10/15/95 09:00	A1	10011007	
2	09/03/95 11:15	B	00110200	
2	10/05/95 13:00	B	00110345	D
3	09/09/95 09:09	C	00120123	

is faster than in the single-table design, designers are free to define foreign keys for current data tables. Although every table has columns that are unique to it, methods such as `describe()` and column attributes such as `dwo.name` enable coding of generic algorithms. An added benefit of the column-duplicated design is that history column information can be displayed without the overhead of an SQL select because it is part of the current data table.

Row-Duplicated Table Design

A row-duplicated table design duplicates the current row in the history table so that the current data table (Table 6) does not have to store the extra history columns (Table 7). This may further speed retrieval of current rows. However, in tables with many current rows, the extra space used by duplicating current rows in the history table may create an unacceptable disk resource constraint. As in the column-duplicated table design, the algorithm to save history rows can be made generic.

The column-duplicated table design was selected for implementation because it is generic across data tables and does not require extra disk storage space. Table 8 compares the strengths and weaknesses of these three table design alternatives and shows that the column-duplicated table design has the most advantages and fewest disadvantages.

Implementation

The algorithm to save history rows for the column-duplicated table design is implemented as a method of a custom user object that contains a datawindow control for current rows and a datastore (a datawindow that has no display attributes, and therefore functions as an SQL-intelligent data array) for history rows. Data members are declared as protected; therefore, access methods are provided. As implemented, numerous methods are included for ancillary functionality such as support for different row selection modes. The history row capability consists of a single public method, `save()`, and a private method for transfer of data that is called in multiple places between tables.

Table 8. Comparison of alternative database table designs for storing history rows.

Design alternative	Strengths	Weaknesses
Single-data table	Less disk space than row-duplicated; generic implementation; fast retrieval of row statistics	Slow retrieval of current rows
Column-duplicated table	Minimum disk space required; fast retrieval of current rows; generic implementation; fast retrieval of row statistics	Union creates slow retrieval of current row plus history rows
Row-duplicated table	Faster retrieval of current rows; generic implementation	More disk space required

With the latest version of the language, the history row portion of the class can be implemented as a separate service class. This more closely matches the instantiation technique of Ada because it can be combined with any datawindow class rather than being coded as part of a specific physical class.

The essential logic for the `save()` method is to loop through modified rows. For each modified row, a row is inserted into the history table datastore. The column values are set to the values in the original buffer of the current table datawindow (the more generic `rowsCopy()` cannot be used because it cannot access original buffers). Then, to service row deletions, the algorithm loops through the delete buffer, and, for each deleted row, inserts two rows into the history table datastore, one for the current row itself and one as a copy of the current row including the current time stamp and user. This latter copy also sets `db_operation_ind` to '`D`'. In contrast to the insert operation, `rowsCopy()` can be used to simplify the insertion of the second copy of the deleted row.

A few rules must be followed to integrate this class into an application. First, the names of parallel fields in the current data datawindow object and history datawindow object must be identical (the table column names can differ because they can be renamed in datawindow objects). Second, the system traces the history of a row via its functional primary key. Therefore, once a functional primary key value is saved, the system must prevent the user from changing the primary key. Otherwise, the old row will be orphaned in the history table with no entry in the current data table. This preventive measure can be accomplished generically with the following statement in the protected attribute of the primary key fields in the current data datawindow: `if(isRowNew(),0,1)`. This statement prevents the primary key from being modified in existing rows. Adhering to these two guidelines ensures that the generic nature of the class is available for any data table.

Results

The model created by Figs. 1 and 4 and Table 2 predicts that applications built from independent, composable components should yield high cohesion, low complexity, high reusability, fewer errors, and lower maintenance cost than normally experienced when not using such components. The average module complexity for custom user objects developed at APL is 10, a value that is within the preferred guidelines.[48] Although it took more time, effort, and knowledge to develop these reusable components, they have already shown cost benefit. For example, no postrelease errors have been reported over the 2-year history of these components. One component that provides security based on organizational structure was especially noteworthy because a system-wide change request required a change in the rules of security. Code modifications in response to the request were made in only a single custom user object that is found in all eight windows affected by the requirements change. This requested change was made months after the release of the application and ran contrary to the original objectives. Our experience at APL underscores the notion that designing for reuse is often beneficial, even if software is not expected to be modified or reused. Making reuse design techniques a standard part of developmental methodology has long-term payoffs.

CONCLUSION

Dramatic gains in software development productivity will be achieved when independent, composable software units are readily available and usable across different environments. The Ada generic formal parameter

is a model for defining highly reusable independent components through composition.

Modern programming languages have features that enable developers to perform or emulate compositional techniques. Understanding the nature and advantages of composable software facilitates successful identification and effective utilization of those features in a language that promotes composition. Some reusable components developed at APL are created using a language feature known as custom user objects, which have many of the features of generic formal parameters. They were available originally only within a proprietary environment but are now available to C++ developers via the Optima++ product by Sybase. Although Optima++ can only be used for Microsoft Windows platforms, it may be extended to UNIX platforms in the future.

Custom user objects are composed of existing independent classes that are standard within the development environment. Coupling is avoided as long as there are no references to global variables or external modules. Cohesion and low complexity are attained through the use of object orientation. Methods are a natural way to decompose functionality, and classes serve to organize them in a formal and convenient manner.

Successful reuse begins at the fundamental level, namely, designing software building blocks to be reusable. This requires greater time and effort but offers long-term benefits. Often, in the early stages of a project, designers and developers are not aware of opportunities or the need for independence or reuse. In such cases, it is tempting to take a quick, nonreusable approach to software development. However, the advantages of components being independent, composable, and reusable often become evident at a later date. Organizations that build and maintain multiple large software systems need to view software as an asset to be managed for long-term gain. By making "design for reuse" the rule rather than the exception, benefits will accrue. Independent, composable components, both those developed in-house and those commercially available, provide the greatest path to reuse.

REFERENCES

[1] Boehm, B., "Improving Software Productivity," *IEEE Software* 4, 43–57 (Sep 1987).

[2] Lanergan, R. G., and Grasso, C. A., "Software Engineering with Reusable Designs and Code," *IEEE Trans. Software Eng.* 10, 498–501 (1984).

[3] Goodell, M., "What Business Programs Do: Recurring Functions in a Sample of Commercial Applications," in *Software Reusability: Volume 2: Applications and Experience*, T. J. Biggerstaff and A. J. Perlis (eds.), Addison-Wesley, New York, pp. 197–212 (1989).

[4] Gruman, G., "Early Reuse Practice Lives up to Its Promise," *IEEE Software* 5, 87–91 (Nov 1988).

[5] Van der Linden, F. J., and Muller, J. K., "Creating Architectures with Building Blocks," *IEEE Software* 12, 51–60 (1995).

[6] Selby, R. W., "Quantitative Studies of Software Reuse," in *Software Reusability: Volume 2: Applications and Experience*, T. J. Biggerstaff and A. J. Perlis (eds.), Addison-Wesley, New York, pp. 213–232 (1989).

[7] Matsumoto, Y., and Ohno, Y., *Japanese Perspectives in Software Engineering*, Addison-Wesley, New York (1989).

[8] Benson, D., "ADGE Systems Architecture Analysis," in *Proc. Domain Analysis Workshop of the Software Productivity Consortium*, Herndon, VA (1991).

[9] Karlsson, E-A., *Software Reuse: A Holistic Approach*, John Wiley & Sons, New York, p. 18 (1995).

[10] Henry, E., and Faller, B., "Large-Scale Industrial Reuse to Reduce Cost and Cycle Time," *IEEE Software* 12, 47–53 (Sep 1995).

[11] Krueger, C. W., "Software Reuse," *ACM Comp. Surv.* 24, 131–183 (Jun 1983).

[12] Prieto-Diaz, R., "Status Report: Software Reusability," *Software* 10, 61–66 (May 1993).

[13] Mili, H., Mili, F., and Mili, A., "Reusing Software: Issues and Research Directions," *IEEE Trans. Software Eng.* 21, 528–561 (1995).

[14] Frakes, W., and Terry, C., "Software Reuse: Metrics and Models," *ACM Comp. Surv.* 28, 415–435 (Jun 1996).

[15] Kliem, R., and Ludin, I., "Making Reuse a Reality," *Software Development* 3, 63–69 (Dec 1995).

[16] Barnes, B. H., and Bollinger, T. B., "Making Reuse Cost-Effective," *IEEE Software* 8, 13–24 (Jan 1991).

[17] Rosen, J. P., "What Orientation Should Ada Objects Take?" *Commun. ACM* 35, 71–76 (1992).

[18] Ambler, S., "Writing Maintainable Object Oriented Applications," *Software Development* 3, 45–53 (Dec 1995).

[19] Booch, G., *Software Engineering with Ada*, Benjamin/Cummings, Menlo Park, CA (1986).

[20] Card, D. N., Page, G. T., and McGarry, F. E., "Criteria for Software Modularization," in *Proc. IEEE Eighth Intl. Conf. Software Eng.* 8, pp. 372–377 (1985).

[21] Kearney, J. K., Sedlmeyer, R. L., Thompson, W. B., Gray, M. A., and Adler, M. A., "Software Complexity Measurement," *Commun. ACM* 29, 1044–1050 (1986).

[22] McCabe, T. J., "A Complexity Measure," *IEEE Trans. Software Eng.* 4, 308–320 (1976).

[23] DeMarco, T., *Structured Analysis and System Specification*, Prentice Hall, Englewood Cliffs, NJ, p. 26 (1979).

[24] Grady, R. B., "Practical Results from Measuring Software Quality," *Commun. ACM* 36, 62–67 (1993).

[25] Schneidewind, N. F., "The State of Software Maintenance," *IEEE Trans. Software Eng.* 13, 303–310 (1987).

[26] Hager, J. A., "Software Cost Reduction Methods in Practice," *IEEE Trans. Software Eng.* 15, 1638–1644 (1989).

[27] Boehm, B. W., "Software and Its Impact: A Quantitative Assessment," *Datamation* 19, 48–59 (May 1993).

[28] Zelkowitz, M. V., "Perspective on Software Engineering," *ACM Comp. Surv.* 10, 197–216 (1978).

[29] Martin, J., and McClure, C., *Structured Techniques: The Basis for CASE*, Prentice Hall, Englewood Cliffs, NJ, p. 563 (1988).

[30] Banker, R. D., Datar, S. M., Kemerer, C. F., and Zweig, D., "Software Complexity and Maintenance Costs," *Commun. ACM* 36, 81–94 (1993).

[31] Card, D. N., Church, V. E., and Agresti, W., "An Empirical Study of Software Design Practices," *IEEE Trans. Software Eng.* 12, 264–271 (1986).

[32] Henry, S., Kafura, D., and Harris, K., "On the Relationship Among Three Metrics," *Perfor. Eval. Rev.*, 81–88 (Spring 1981).

[33] Kafura, D., and Reddy, G. R., "The Use of Software Complexity Metrics in Software Maintenance," *IEEE Trans. Software Eng.* 13, 335–343 (1987).

[34] Ward, W. T., "Software Defect Prevention Using McCabe's Complexity Metric," *Hewlett-Packard J.*, 64–69 (Apr 1989).

[35] Troy, D. A., and Zweben, S. H., "Measuring the Quality of Structured Designs," *J. Syst. Software* 2, 113–120 (1981).

[36] Basili, V. R., and Perricone, B. T., "Software Errors and Complexity: An Empirical Investigation," *Commun. ACM* 27, 42–52 (1984).

[37] Shen, V. Y., Yu, T., Thebaut, S. M., and Paulsen, L. R., "Identifying Error-Prone Software—An Empirical Study," *IEEE Trans. Software Eng.* 11, 317–323 (1985).

[38] Weiss, D. M., and Basili, V. R., "Evaluating Software Development by Analysis of Changes," *IEEE Trans. Software Eng.* 11, 157–168 (1985).

[39] Selby, R. W., and Basili, V. R., "Analyzing Error-Prone System Structure," *IEEE Trans. Software Eng.* 17, 141–152 (1991).

[40] Bitman, W. R., "Information Systems Modeling: An Object Oriented Development Method," in *Proc. Washington Ada Symp.* 9, Reston, VA, pp. 93–105 (1992).

[41] Parnas, D. L., "On Criteria to be Used in Decomposing Systems into Modules," *Commun. ACM* 15, 1053–1058 (1972).

[42] Parnas, D. L., and Siewiorek, D. P., "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems," *Commun. ACM* 18, 401–408 (1975).

[43] Parnas, D. L., Clements, P. C., and Weiss, D. M., "The Modular Structure of Complex Systems," *IEEE Trans. Software Eng.* 11, 259–266 (1985).

[44] Bitman, W. R., "Complexity Metrics as Feedback Mechanism for Object Oriented Software Engineering," in *Proc. Intl. Conf. Industrial and Applied Mathematics* 2, Washington, DC, p. 20 (1991).

[45] Bitman, W. R., "Functional Lists: Object Oriented Design Classes for MIS Applications," in *Proc. Washington Ada Symp.* 7, Reston, VA, pp. 101–122 (1990).
[46] Ichbiah, J. D., Heliard, J. C., Roubine, O., Barnes, J. G. P., Krieg-Brueckner, B., and Wichmann, B. A., "Rationale for the Design of the Ada Programming Language," *SIGPLAN Notices* 14 (Jun 1979).
[47] Bott, M. F., and Wallis, P. J. L., "Ada and Software Reuse," in *Software Reuse with Ada*, R. J. Gautier and P. J. L. Wallis (eds.), Peter Peregrinus, Ltd., London, pp. 3–13 (1990).
[48] McCabe, T. J., and Butler, C. W., "Design Complexity Measurement and Testing," *Commun. ACM* 32, 1415–1425 (1989).

ACKNOWLEDGMENT: I gratefully acknowledge the help of Laura M. Ellerbrake and David O. Harper for reviewing the material and for adding insights to the model.

THE AUTHOR

WILLIAM R. BITMAN received a B.A. in biology from Yeshiva University (New York) in 1972 and an M.S. in genetics from Boston College in 1975. In 1994, he was awarded a Master of General Administration in Management Information Systems degree from University College, University of Maryland. He joined APL in 1993 as a resident subcontractor in the Computer Applications Group and accepted a position as a member of the Senior Professional Staff in 1994. Mr. Bitman has given presentations at the Washington Ada Symposium, ACM Computer Science Conference, International Conference on Industrial and Applied Mathematics, and International PowerSoft Users Conference. He is a member of ACM SIGAda and served as vice president of the Baltimore chapter in 1988. Currently, he is vice president of the Baltimore Area PowerBuilder Users Group. His e-mail address is William.Bitman@jhuapl.edu.