Forging a silver bullet from the essence of software

by R. G. Mays

Most improvements in software development technology have occurred by eliminating the accidental aspects of the technology. Further progress now depends on addressing the essence of software. Fred Brooks has characterized the essence of software as a complex construct of interlocking concepts. He concludes that no silver bullet will magically reduce the essential conceptual complexity of software. This paper expands on Brooks's definition to lay a foundation for forging a possible silver bullet. Discussed are the three essential attributes of software entities from which a number of consequences arise in software development: (1) conceptual content, (2) representation, and (3) multiple subdomains. Four basic approaches to develop technologies are proposed that directly address the essential attributes. Although some of these technologies require additional development or testing, they present the most promise for forging a silver bullet. Among them, design reabstraction addresses the most difficult attribute, multiple subdomains, and the most difficult consequence, enhancing existing code, making it the best prospect.

In his paper on "no silver bullet,"[1] Fred Brooks addresses the problem of improving the quality and productivity of software development. Brooks contends that most of the improvements in software development technology in the past have occurred by eliminating the accidental or nonessential aspects of the technology. For example, the use of high-level languages removed much of the incidental complexity associated with the hardware (internal data representations, registers, peripheral interfaces, etc.). Time sharing removed the incidental problems associated with the need to compile programs in batch, thus losing the immediacy of our thinking in the activity of programming.

Further progress now depends on addressing the essence of software. After reviewing a number of promising approaches, Brooks concludes that there is no silver bullet that will by itself magically reduce the essential conceptual complexity of software and allow an order of magnitude improvement in productivity and quality.

Brooks's arguments and conclusion, presented nearly seven years ago, captured the imagination of the software development community and have had a continuing influence on its members' thinking. Other authors have recently proposed additional technologies for consideration, for example, design execution[2] and knowledge-based software engineering.[3]

Brooks has provided a yardstick for technological breakthroughs, namely a tenfold improvement in quality or productivity, or both. Such a yardstick has engendered a level of realism when considering technological improvements. A given technology may not be a silver bullet. Therefore, we should not expect, nor should providers imply, wildly dramatic results. Rather, we should go

with what improvements can be obtained. The combined effect of several of these improvements may well be an order of magnitude improvement.

More importantly, Brooks's arguments have established a framework for thinking about the nature of software and of programming: there are

If a silver bullet can improve software development, it can only be forged by addressing the essential nature of software.

essential aspects of software that must be addressed by any approach to improve the technology.

Brooks's paper in many ways echoes the views set forth by Parnas[4] in a series of minipapers published a year earlier. In them, Parnas characterizes software systems as "discrete state" systems without repetitive structures and having a large number of states. For Parnas, the complexity of the software system is the result of the very large number of states contained in it, which prohibit the developer from obtaining a complete understanding of the behavior of the system.

In this paper, I propose to examine the ideas presented by both Brooks and Parnas in more detail. If a "silver bullet" can possibly improve software development, it can only be forged[5] by directly addressing the essential nature of software. Thus, a more detailed look at the essence of software is warranted, which we hope will suggest approaches for dealing with the essential complexity of software. Even if an order of magnitude improvement in quality and productivity cannot be achieved, these approaches will constitute the most profitable and possibly the only effective attack on the problem.

What constitutes the essence of software?

What would the essence of software be like? The *essence* of a thing is that which gives it its identity. It is the inherent, unchanging nature of the thing. Essential attributes are those properties that are intrinsic and indispensable, as opposed to coincidental or *accidental.*

Thus, for those of us who are intimately familiar with software, the essence of software ought to be obvious: we should be able to recognize it instantly. The essence ought to be simple and straightforward, rather than abstruse. And the essential nature should be ubiquitous: if it is truly essential, it should apply to all forms of software, regardless of the specific problem domain, application, or language representation.

Given this basic information, how do we characterize the essence of software? Brooks calls the essence of a software entity "a construct of interlocking concepts," consisting of such things as data sets, data items, algorithms, and functions. The conceptual construct is abstract, highly precise, and richly detailed.

Brooks distinguishes the conceptual construct itself, which is abstract, from its concrete representation in some programming language or other. The conceptual construct can be represented in any number of languages. The difficulty, Brooks asserts, is more in specifying, designing, and testing the conceptual construct than in coding it in an implementation language.

Parnas views software systems as "discrete state" systems with a large number of states. The large number of states results from the fact that software systems do not have repetitive structures (which would reduce the overall system complexity) such as are found in computer hardware, for example. The number of states increases because of the conditional nature of the execution of the program: whatever the program does at any given point depends on what happened in the past. Each step of the program gives rise to additional states on which subsequent steps depend, and so on.

I propose to take Brooks's definition of the essence and Parnas's notion of discrete states and expand on these ideas. The essence of a software entity is *a construct of interlocking concepts* with the following essential attributes:

1. Conceptual content: A software entity is characterized by concepts that come from both the

problem domain and the surrounding software entities with which it interfaces.
2. Representation: The concepts of a software entity are expressed as representations of both the data it uses and functions it performs.
3. Multiple subdomains: A software entity performs functions that consist of transformations on its data, based on conditions present at the time of execution. The presence of conditions splits the input domain into multiple subdomains of function.

Conceptual content. All software contains concepts that come from its *problem domain*, that is, the domain of operation in which the function of the software has a useful purpose. For example, the problem domain of a payroll system includes employee time records, payroll payments, payroll statements, tax withholdings, tax reporting, tax statements, etc. The problem domain of an automobile microprocessor includes engine operating status, engine environment conditions, engine controls, dashboard displays, etc. The problem domain concepts are the same as the concepts stemming from our ordinary human experience.

In addition, all software contains concepts from the *surrounding software entities* with which it interfaces. The software uses the constructs of the lower-level entities and in turn is used by higher-level entities, an end user, or an external interface. We can call this area of interaction the environment, or *milieu*, of the software. For example, a component of an airline reservation system uses a database subsystem. Thus its milieu includes the concepts of the database interface: database records, keys and fields, sequential access, access by index key, etc.

As a result, a hierarchy of software components develops having a unique conceptual content at each level on which still higher components are based. At the lowest level, the software interacts directly with hardware features (e.g., channels, control registers, hardware interrupts, device buffers, etc.) and thus reflects the concepts of those features. Higher software levels are built upon the lower levels and use the concepts from those levels (e.g., block I/O, device protocols, display I/O, etc.). Still higher levels use the concepts from the levels below them. Thus, a large network of interrelations is built up from the concepts operant at each level of software. At the highest levels, the problem domain concepts are operant.

Within any given level only the concepts that are operant at that level have significance. The concepts of the other levels, for example, those from

Concepts of a software entity are expressed as representations in both the data it uses and the functions it performs.

levels that are two or more removed, are hidden and are not part of the milieu of the software at that level.

Part of the milieu of a software entity is the programming language in which it is written, which includes concepts of the syntactic structures (e.g., IF-THEN-ELSE, DO WHILE-END), built-in functions (e.g., SUBSTR, TRANSLATE), the scope of variables, etc.

Representation. The concepts of a software entity are expressed as representations in both the *data* it uses and the *functions* it performs.[6] Data usually have associated with them a name (e.g., GrossPay) and a specific value (e.g., 2715.50). This applies to simple data items, aggregates such as arrays and structures, abstract data types, and objects.

The concept of a data item is suggested by its name. Frequently, however, more explanation is needed to specify the details of the data item, such as the detailed definition, its usage in different contexts, the meanings of different values, etc. The value of a data item is the instantiation of the concept in a specific context (e.g., 2715.50 is the gross pay for employee number 143300).

The functions of a software entity usually have associated with them a name (e.g., SetEmployeeSalaryData) and other information about the function performed (e.g., required input variables, functions performed, possible output var-

iables, and expected values). Functions can be viewed as a transformation of input data values to output data values provided by one or more series of steps of the program (e.g., for a given EmployeeNumber the steps of the program set various employee salary data items, such as GrossPay, Exemptions, FilingStatus).

The concept of the function is suggested by its name, but the details of the transformation that the function performs are contained in other information and specifications of the function. Functions can be *encapsulated*, that is, defined within a syntactic structure, such as a procedure, function, or method definition, which may have associated formal parameters that define the interface. The encapsulating structure separates those specific program steps from other parts of the program that use those steps.

The representations in a software program have a correspondence with the concepts for which they stand. A variable named NetPay, for example, is not a quantity of money; rather, its value *represents* a quantity of money, which is the employee's net pay. A function named ComputeDeductions, for example, *represents* those steps of the program that take the representations for an employee's gross pay, exemptions, filing status, and so on and produce representations for federal income tax, social security tax, state tax, and other deductions. The function ComputeDeductions thus represents the steps of the program that implement the concept of the function, in this case, computing payroll deductions.

In evaluating the representations of data and function, we can speak of their *conceptual integrity*, that is, how well the representation matches the concept behind it. The idea of conceptual integrity is an extension of the notion of the *cohesion* of a module that deals with how well-related the functions of the module are to one another, in other words, how well they relate to the overall concept of the function of the module. The idea of conceptual integrity also applies to the relatedness of the representation of data structures and other functional structures such as object classes. Conceptual integrity is important when dealing with the representations of a system over time as modifications and enhancements are made to the system.

Multiple subdomains. A software entity performs functions that consist of transformations on its data, based on conditions that are present at the time of execution. That is, what the software will do in any given instance of execution depends on the conditions that are present as, for example, in data values in memory, console switch settings, and similar indications of the state of the computer. Depending on these conditions, the software will transform its data values in a specific way.

Representations in a program have a correspondence with the concepts for which they stand.

Thus, a software entity will have any number of different, discrete effects on its data depending on what conditions are present. A particular program fragment might, for example, compute the square root of a number in all instances where the input number is positive or zero, and set an error indicator when the input number is negative. The different conditions that govern the behavior of the software can be viewed as those sets of input data values that cause a different type of transformation. In the example given, the set of negative numbers and the set of nonnegative numbers give rise to different behaviors in the program fragment.

The different sets of input conditions that define the different behaviors of an entity are called the *subdomains of the input domain*. With inputs in one subdomain, the software will have one behavior, whereas inputs in another subdomain will result in a different behavior. The process of dividing the input domain into its subdomains is called *partitioning the input domain* (see References 7 and 8).

The notion that software has multiple subdomains is equivalent to Parnas's notion of multiple states and to the idea of multiple paths through a program fragment. The concept of multiple subdo-

mains has the advantages of (1) the functional concept of a subdomain can be more readily identified, (2) the subdomains in most cases can be more readily derived from the syntax of the code,

Multiple subdomains are an intrinsic property of all software.

and (3) implicit conditions can be handled. For the purpose of this paper, however, the reader can substitute the concepts of multiple states or multiple paths.

The property of multiple subdomains arises with the use of programming language constructs that cause conditional execution. The best example of this situation is the IF-THEN-ELSE construct that is present in some form or other in most languages:

```
if condition-A then do
   transform-X
end
else transform-Y
```

If condition-A is true, then transform-X is performed. If condition-A is not true, transform-Y is performed. The IF-THEN-ELSE construct creates two distinct paths of execution for the program, depending on condition-A at the time of the execution.

The input domain of this program fragment has two possibilities (i.e., condition-A and *not* condition-A) that give rise to two distinct possible outcomes (i.e., transform-X or transform-Y). Another notation for this IF-THEN-ELSE construct that expresses the partitioning of the input domain into its two subdomains might be:

```
1. condition-A
     → transform-X

2. #NOT# condition-A
     → transform-Y
```

All language statements create at least one subdomain, namely the function of that statement. However, many constructs create multiple subdomains. For example, the SELECT-END construct creates a subdomain for each nested WHEN and OTHERWISE statement. Some constructs have *implicit subdomains*. For example, the sequential read statement in many languages implicitly raises a condition or sets a return code when end-of-file occurs. In these languages the sequential read statement has two subdomains, the successful read and the end-of-file condition.

When statements follow one another sequentially in a program, the subdomains of the combined statements can be derived by combining the subdomains of each sequential statement. In forming this combination, the different conditions are "multiplied" together in their different permutations, and the transformations are combined sequentially. Similarly, the subdomains of nested statements (e.g., statements within a DO-END group) are combined in the same way with the subdomains of the nesting statements. A subroutine call or similar construct acts as a single statement with multiple subdomains inherited from the called routine.[9]

The property of multiple subdomains, each with a set of conditions and transformations, arises from the conditional results that occur in the execution of various language constructs. However, the conditional results are the same regardless of the language in which the software is written. The implementation of equivalent functions in different languages has the same subdomains even though different language constructs are used.

Multiple subdomains are thus an intrinsic property of all software, regardless of language representation. All software entities have multiple subdomains in which a different transformation will occur, depending on which sets of conditions in the input domain are present during execution. In other words, the behavior of all software can be expressed abstractly in behavior specifications of the form: under conditions A, transformation X will occur; under conditions B, transformation Y will occur; under conditions C, transformation Z will occur, and so on.

The essence of software. Thus, a software entity is in essence a construct of interlocking concepts characterized by a *conceptual content* derived from its problem domain and the milieu of other software entities with which it interfaces, by *rep-*

resentations of its concepts both in the data it uses and in the functions it performs, and by the *multiple subdomains* of its input domain that characterize the different transformations that will occur, depending on the conditions that are present during execution.

Consequences of three essential attributes of software

A number of *consequences* necessarily follow from the three essential attributes of software. Because software has certain inherent attributes, other conditions and properties follow. From these consequences, still other consequences arise. The consequences thus further elaborate what is basic to software. Moreover, they influence the nature of software development and define the bounds within which software technology can be effective.

I wish to propose 10 consequences, shown in the diagram in Figure 1 with their logical connections. For example, that programs are objects in the world is a consequence of the fact that they are a "crystallization" or encoding of the conceptual construct of the software, which in turn is a consequence of the essential attribute of representation. Of course, these consequences are not the only ones that may be proposed. Further, the connections shown in the diagram are the most direct connections but clearly there are other relations between these consequences.

In his paper, Brooks cited four inherent properties that arise from the essential nature of software: complexity, conformability, changeability, and invisibility. These same inherent properties also arise from the 10 consequences I am proposing. A fifth inherent property, namely that software is developed primarily through incremental enhancements, results as well.

The conceptual construct of the software is held in the developer's thinking (1). When we specify and design a software entity, we define the conceptual construct (i.e., its key concepts) and refine those concepts into ever more detailed data and functional representations until finally the code is produced. The role of the developer's thinking in this design process has largely been ignored, yet it is the primary faculty we use in developing software.

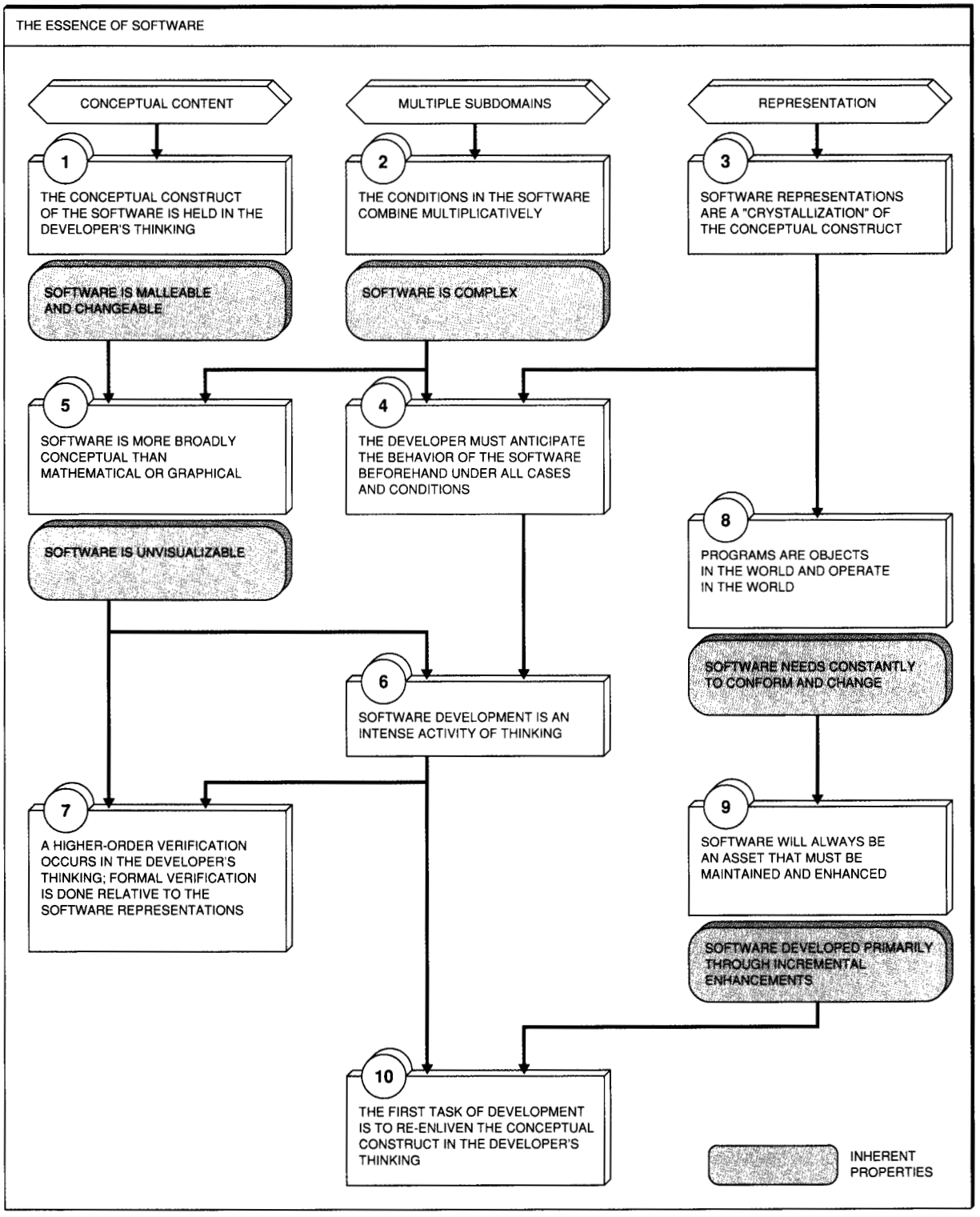
As developers, we develop the concepts of the system and represent them in specifications and designs. Our thinking is constantly engaged as we mull over the details in our thoughts. We may make notes and draw preliminary design notations on paper or a blackboard. We think through the concepts in detail before putting them into a formal representation. Once we have a design written down we bring it up again and again in our thinking, reviewing details, exploring the implications of new concepts, refining our conceptual constructs, and finally recording the refinements and changes in the representation.

We develop and refine the concepts first in our thinking and then record the result in a design representation. We produce ever more detailed representations and finally the implementation. During this process, the conceptual construct or *conceptual view* is active in our thinking and has a *living* quality. If we are absorbed in this process, our thinking activity even appears to continue in us unconsciously during periods of relaxation and sleep, as indicated by many developers who report that insights about the system come to them after they wake up or during times when they are relaxing.

The written representations we develop of the design serve two purposes: as an aid to recall the conceptual view and to direct our attention to one aspect or other of it, and to communicate the conceptual view to others so that they can include it in their own thinking. A good design representation accomplishes these two purposes readily.

Once we are done with the development effort, the conceptual constructs that we developed in our thinking begin to fade. No longer can we as readily recall the conceptual view of the design. If we must go back to the design or code to fix a bug or develop an enhancement, for example, we must reconstruct the conceptual view in our thinking. This process becomes one of rediscovery: as we look at the code and whatever design documentation exists, various concepts are recalled from memory, and we gradually build up the conceptual view. If we are looking at someone else's code, the process is one of pure discovery of the conceptual view. We must discover and build up all of the details of the conceptual view from scratch. We use whatever information is at hand: the code, the names of data items, the contents of data files, reports and screens, comments

Figure 1 Consequences of the essential attributes of software



in the code, and any other documentation that is available.

In the process of software design, we can speak of the developer's *clarity of thinking*. By this we mean how clearly defined the concept is in our thinking, how "concrete" it is, how familiar we are with it, how easily we can consider different aspects of it, how readily we detect inconsistencies with it in what is presented to us by others or what we come across in our design work.

When the conceptual view of the system is active in our thinking, it takes on a fluid character. We can change it as readily as we can change the concepts in our thinking. In this way, software is inherently malleable. Its ease of change arises from its conceptual nature: since the structure of software is purely conceptual, "pure thought-stuff" as Brooks puts it, change is accomplished by simply changing the conceptual structure. Software is malleable because it is purely conceptual: there are no physical elements whose properties and limitations the software engineer must take into account.

The conditions in the program combine multiplicatively (2). The subdomains of each part of a software entity combine together to form a set of subdomains for the entity as a whole. The subdomains result from multiple internal conditions and similar interactions in the interfaces in the milieu of the software. In general, the subdomains of any two software segments that interact with one another (whether by sequential execution, by nesting, by looping, or by a call interface) are the cross product of the subdomains of each segment.

Typically, the set of subdomains for a program is very large because the interactions of the conditions in each segment are always multiplicative. Take for example a 10 000-statement program in which every fifth statement splits the domain with some sort of condition. Then the program will have 2000 conditional statements and could have anywhere from 2001 to 2^{2000} (or about $2x10^3$ to 10^{600}) subdomains. (See the Appendix for a derivation of these relationships.)

Moreover, the cases that favor the lower estimate in this range (nested conditional statements) are typically less likely than the cases that favor the higher estimate (sequential conditional statements). Thus, the number of subdomains of a 10 000-statement program, a "small" program by most standards, can be literally astronomical.

The inherent property of complexity noted by both Brooks and Parnas is a direct consequence of the multiple subdomains resulting from the conditional tests in the code. Each subdomain is equivalent to a distinct state that the software entity can be in. Each subdomain arises from a conditional structure, such as an IF-THEN-ELSE, but has a conceptual aspect as well. The interaction of multiple conditions together causes a complex conceptual expression of conditions for the subdomain. The transformation of the subdomain likewise can be a complex sequence of transformations that accomplish a unique overall function or outcome.

The subdomains of each part of a software entity combine to form a set of subdomains for the entity as a whole.

For Parnas, the use of structured methods (whereby the software function is encapsulated in smaller, simpler modules) does not reduce the complexity of the entity sufficiently because even then complex interactions remain between the encapsulated components. The use of mathematical logic will likely not help because the mathematical expressions themselves are extremely complex. The complexity of software is beyond the capacity of the human mind to comprehend.

Software representations are a "crystallization" of the conceptual construct (3). As we develop the conceptual view of the software entity, we begin to record representations of the concepts in some form as specifications and designs. We continue to refine these representations as our thinking progresses and as changes are considered and decided upon. The written representations come to embody the proposed software in greater and greater detail, until finally the software is coded.

In contrast to the fluid character of our thinking, the representations are fixed and rigid. They can be viewed as a "crystallization" of our thought activity, an embodiment of our thinking that serves to remind us of what we have thought previously, much as the writing on a page calls up in us the thoughts that were present when it was written.

However, because they are fixed, the representations become less malleable. The more detail we have set down, the more difficult and error prone it becomes to change them. Ultimately, when the software is coded, it becomes an object in the world. Thus, we have transformed what began as pure thought into a fixed object that operates in the world.

The developer must anticipate the behavior of the software beforehand under all cases and conditions (4). Because of its fixed nature as a crystallization of the conceptual construct, the software entity is inflexible in responding to inputs in any way other than the one in which it was programmed. Programs will perform only within the limits of what the developer intended and had the foresight to include. Because of this inflexibility, the developer must anticipate and explicitly handle all possible outcomes and responses of the system. This fact in turn gives rise to numerous conditions in the code, resulting in a very detailed, complex construct. As Brooks puts it, the software entity is "highly precise and richly detailed."

Conditions occurring in the operation of the software that were *not* anticipated by the developer result in either a limitation on the operation of the software or an actual defect in its operation. The developer must then either alter and extend the function of the software or fix its operation under the unanticipated condition.

Software is more broadly conceptual than mathematical or graphical (5). A consequence of the purely conceptual nature of software and its complexity is that the use of graphical or visual methods and the use of mathematical methods have limitations. Let us consider each of these in turn.

Graphical or visual methods. Brooks contends that software is invisible and unvisualizable. Software cannot be visualized because geometric or spatial representations are inadequate to represent its complex relationships. Of course, it is possible to produce graphs of relationships for flow of control, flow of data, patterns of dependency, time sequence, and name-space relationships. However, we can only represent these relationships in a simple presentation (e.g., hierarchical) if we hide or cut links.

The software entity is inflexible in responding to inputs in any way other than the one in which it was programmed.

Part of the unvisualizability of software arises from its complexity as described earlier. The other part arises from its conceptual content. It is difficult to represent concepts generally, from all possible problem domains, except through textual descriptions. The concepts themselves in general are not spatial or embedded in space. Even David Harel, a strong proponent of visualization in software development, concedes that the algorithmic operations of software will probably remain textual.[2] It is the *structural relationships* between the constructs of a software entity (e.g., calling structures, data relationships) that lend themselves to graphical representation, as opposed to the essential conceptual content.

This is not to say that graphical and other visual methods are not useful in working with software entities. On the contrary, they are quite useful. However, they do not fully address the conceptual nature of software.

Mathematical methods. A widely held view in computer science (for example, References 10 and 11) is that programs are mathematical objects. According to this view programs can be treated as functions operating over their domain of input, which can be expressed as formulas in discrete mathematics describing the transformation of input data into output data. In taking advantage of its mathematical nature we can use simplifying concepts to prove the correctness of a program. Programs can be reasoned about with

mathematical rigor. A mathematical treatment of programs helps deal with their complexity, bringing the developer "intellectual control" and mastery over the complexity. Take for example the following statement from E. W. Dijkstra:[10]

> As soon as programming emerges as a battle against unmastered complexity, it is quite natural that one turns to that mental discipline whose main purpose has been for centuries to apply effective structuring to otherwise unmastered complexity. That mental discipline is more or less familiar to all of us, it is called Mathematics. If we take the existence of the impressive body of Mathematics as the experimental evidence for the opinion that for the human mind the mathematical method is indeed the most effective way to come to grips with complexity, we have no choice any longer: we should reshape our field of programming in such a way that, the mathematician's methods become equally applicable to our programming problems, for there are no other means.

Although many hold the view that programs are mathematical objects, the presumption that the mathematical treatment of programs simplifies or reduces complexity is not as universally held. Parnas holds that the mathematical functions describing the behavior of programs are not continuous functions. Neither traditional engineering mathematics nor other mathematical methods, such as formal methods and verification, will help overcome the inherent complexity of software. He asserts that "The large number of states and lack of regularity in the software result in extremely complex mathematical expressions. Disciplined use of these expressions is beyond the computational capacity of both the human programmer and current computer systems."[4]

Brooks points out the premise behind the expectation that mathematics will reduce complexity, namely that mathematics has permitted a reduction in complexity in the physical sciences because there are underlying unifying principles which in themselves are simple. However, there are no such underlying unifying principles in software: the complexity is there because of the arbitrary complexity of human institutions to which the software must conform.

As we have seen, the complexity of software as measured by its subdomains results from its conditional nature. As a condition is added to the program, the number of subdomains can increase either by one or by double its current number. No unifying principles are involved in software because the conditions that are tested are unique to the particular aspect of the problem being handled at that point in the program. No simplifying principles will significantly reduce the number of subdomains, if we assume that the normal design practice of factoring out common function into subroutines or similar structures has been followed.

Thus, the mathematical expression of the function of a program fragment contains the *same complexity* as the software itself because it must express the function of each of the subdomains of the fragment. Yes, there are some instances where a mathematical expression can hide the subdomains in a program fragment, as for example:

```
if x > y then a = x
         else a = y
```

This statement contains two subdomains but can be replaced with a single functional expression containing only one subdomain:

```
a = MAX(x,y)
```

However, such instances tend to be confined to the use of mathematical expressions that happen not to be available in the implementation language. Such mathematical expressions are simpler than the program code but are hardly an overall simplifying principle that can be applied in all or even in many instances to reduce complexity.

The use of mathematical notation also attempts to summarize the function of a program fragment into a single transformation, that is, to take a procedural expression of the function over many program statements and express it in a single "instantaneous," or one-step, function. Although this process can result in a smaller expression of the function, the expression is rarely less complex. In some instances, the mathematical notation may actually obscure the meaning of the function because it focuses on the mathematical structure of the transformation rather than the concept behind it. For example, a specialized sort routine may require a very complex mathematical structure to describe its function. The concept

inherent in the logic can be simply stated conceptually as "sort using the xyz technique."

Because software entities are conceptual constructs, it is more appropriate to view them as *conceptual* objects rather than more restrictively as mathematical objects. The mathematical treatment of programs is one way of working with them as conceptual objects because mathematical concepts are a subset of all concepts. However, the mathematical treatment of a program is fundamentally limited in that it cannot abstract the function of the program beyond a certain point and still remain mathematical.

Any abstraction of a function in mathematical terms (mapping inputs to outputs via a functional transformation) cannot be further abstracted without resorting to higher or broader concepts that do not explicitly state the functional transformation. Any abstraction of the partition of the input domain in mathematical terms (by listing the subdomains with their conditions and transformations) cannot be further abstracted without resorting to higher concepts that do not explicitly state the subdomains.

For example, the mathematical abstraction of a complex search function can be expressed in mathematical transformations that apply across each of its subdomains. However, the next higher abstraction of this function can be expressed only in terms like "search the database for records satisfying the criteria," a *nonmathematical* conceptual abstraction that encompasses the details of both the transformation and the subdomains of the function.

The most common objection to this further step of abstraction beyond the mathematical expression is loss of its precision. The terms "search" and "satisfying the criteria" may be ambiguous and therefore are unreliable. In addition, Brooks asserts "Descriptions of a software entity that abstract away its complexity often abstract away its essence."[1]

I disagree with both objections. To begin with, the higher abstraction in which we choose to express the function of the program fragment is the very concept that motivated us to construct the function in the first place. The precision of expression and degree of ambiguity that we had at design time was no more or less. Yet from that concept we developed the detailed implementation. If the concept (search for records satisfying the criteria, in the example) was precise enough and clear enough at that level, it should still serve us as an adequate abstraction *at that level*. The concept is still valid at its level of abstraction. If we need further clarification or precision, we can simply look at the lower levels of abstraction (the mathematical expression or the code itself).

Furthermore, abstraction using ever broader concepts that encompass more and more parts of the software entity does not "abstract away" its essence because its essence consists of these very concepts, provided they are the right concepts, of course. The complexity of the entity arises from the details at ever lower levels of abstraction. So long as these details are readily accessible, nothing is lost to the developer. The concepts at each level of abstraction are available to the developer to take up in his or her thinking activity. The key is to abstract the function using concepts with sufficient detail so that the abstraction truly encompasses the function.

The use of mathematical expressions and proofs or tests of equivalence are useful for refinement of designs and for verification of the implementation. Though mathematical treatment of a software entity provides precise, unambiguous expression of the function of the entity, it does not help remove complexity. Only the use of broader, nonmathematical conceptual abstractions can help the developer to encapsulate the complexity of the software entity.

Software development is an intense activity of thinking (6). Because of the conceptual character of software, software development is fundamentally an activity of thinking. Because of the complex conditional nature of software and the need to anticipate the behavior of the software in all circumstances, under all cases and conditions, the developer's thinking must be precise and thorough. Because of the purely conceptual nature of the software, graphical or visual and mathematical methods are limited in the degree to which they can assist the process of development.

Thus the software developer must rely on his or her thinking activity as the primary means of developing the conceptual construct. Any experienced developer will confirm the intense mental activity that is involved. We become absorbed in

the conceptual view, in some instances almost consumed in the intensity of thinking.

A higher-order verification occurs in the developer's thinking; formal verification is done relative to the software representations (7). Verification is the process of comparing a lower-level design representation or the code for a software entity against its higher-level design representation to determine how well it matches. Verification frequently refers to a formal process comparing the specifications for a program fragment and its corresponding design or code. A very common complaint (see Parnas,[4] for example) is that the specifications themselves may contain errors and thus are not a reliable representation of the system.

Once coded, programs become objects in the world.

Specifications are expressed at a higher level, usually giving only the expected external behavior of the system. They allow developers to specify the behavior of the system independent of and prior to its implementation. They serve a useful function in forcing a more detailed definition of the conceptual construct. They can thus serve both as an individual reference for the developer and for communication among developers.

The specifications and the design have a relationship with a third element, namely the developers' thinking. The developers' thinking really precedes the writing of the specifications and the design. It is the common ground from which they are both derived. Both the specifications and the design embody the same conceptual view because the same concepts and ideas have motivated each.

With this three-way relationship in mind, it is no wonder that a number of things can go wrong: the developers' conceptual view may be flawed; one or more developers may have a view that differs from the others'; the specifications may reflect these flaws or may have errors that misrepresent the concepts; the designs and code likewise may contain the conceptual flaws or other errors. Therefore, the process of verification against specifications will catch only some of the errors because the verification is essentially a mechanical process of comparing the consistency of one representation with another.

We should realize that flaws in our thinking are a natural outcome of how we work: we may uncover significant problems in our conceptual view as we learn more about the problem domain or see more clearly the implications of our current thinking. In one project I was involved with, we called these flaws in our thinking "major conceptual errors," or MCEs. We were uncovering MCEs still fairly late in the design process.

Because our thinking is involved, no formal verification is possible that will detect flaws in our conceptual view. However, because our thinking is involved, we should realize that a *higher-order verification* is always going on in which we compare the information at hand with our conceptual view of the design. We verify everything we encounter about the system relative to our understanding of the problem domain. We apply whatever knowledge we have, including our commonsense understanding of the world. Thus, the "real" verification occurs in our thinking, throughout the development process, even as we perform, for example, the formal verification of the design relative to the specifications. Robert Glass[12] makes a similar argument that software design is in essence a cognitive process that is not susceptible to formal approaches.

Programs are objects in the world and operate in the world (8). Once they are coded, programs become objects in the world, that is, "things" we can use, sell, teach about, maintain, enhance, etc. We also come to rely on them as tools to do our work. The software itself is indestructible, that is, it does not deteriorate or wear out over time. However, because it is fixed, a crystallization of our thinking at some point in time, it may become obsolete. As time goes on, conditions in the world change, and the software no longer quite "fits."

Thus a software entity needs constantly to conform and change to its changing environment,

that is, to human institutions, practices, and human interfaces. Program modifications are frequent because the software of a system embodies its function, and most pressure for change occurs in its function. Also, software is the preferred element of a system to change because it is malleable and more easily changed compared to other system elements such as the hardware.

Most software development efforts involve creating incremental enhancements to existing code.

The property of conformity arises directly from the conditions imposed by the external requirements and interfaces. For example, the requirements for U.S. social security tax calculations have changed several times in the last decade. Not only has the tax rate changed practically every year, but the basic structure of the tax changed from a single tax rate for both employers and employees, to a different rate for employers, back to a single rate, and more recently to a separate tax schedule and separate reporting for social security tax and for medicare tax. Each of these changes required that payroll software conform to the new requirements via new conditions in the code for separate tax rates, a separate tax calculation, and separate reporting of the taxes.

Software will always be an asset that must be maintained and enhanced (9). Because programs are objects in the world in which we have invested time and effort, they become assets that we feel obligated to maintain and enhance. We cannot afford to rewrite software every time we must alter it. In fact, only when there is great justification to rewrite, for example, when the existing or legacy code can no longer be maintained or enhanced, do we undertake to completely replace the program. Even when we do decide to re-engineer an existing system, the resulting new system becomes new legacy code that we then feel obliged to maintain and enhance.

Thus, most software development efforts, in applications software, systems software, and imbedded (microcode) software, involve creating incremental enhancements to the existing, legacy code. New functions and modifications to existing functions are added to the existing software base. Seldom do we develop completely new code for a new product or system. Many so-called "new systems" in fact turn out to be based on existing code, code that may have been developed as a prototype, an internal tool, a field-developed program, etc., and as a result many of these "new systems" involve enhancing existing code.

This fact of life of software development is confirmed by industry estimates. The proportion of development effort devoted to software enhancements, as opposed to totally new code, is usually estimated at 80 percent or more.[13] Despite the overwhelming proportion of effort devoted to enhancing existing code, most software development technologies address only the new development paradigm and thus are not readily applicable to legacy software enhancements. This mismatch of technology to application is an apparent blind spot in this industry.

The first task of development is to re-enliven the conceptual construct in the developer's thinking (10). When the software developer is faced with maintaining or enhancing an existing program, the first task is to discover the relevant parts of the conceptual construct that are needed for the change. This activity is the opposite of the original task of setting down the conceptual construct into design representations and code. It involves looking through the code (and possibly other documentation) and *re-enlivening* the conceptual construct in our thinking. For code that we have written ourselves, this activity becomes a process of rediscovery and recalling from memory the conceptual view we had earlier. For code that we have not written, it is a process of discovery of the conceptual view from scratch, using whatever conceptual information is contained in the code or other documentation.

Only when we have the conceptual construct active again in our thinking can we properly develop the incremental enhancement. We do not typically need to recapture the entire conceptual construct (nor for large software systems can we) but

only those parts of the design that are relevant to the specific enhancement.[14]

The process of recapturing and re-enlivening the conceptual view uses *program understanding* and *design reabstraction*. Program understanding is the process of understanding what exists in the code. Design reabstraction is the process of abstracting the higher-level concepts from the code, possibly building design representations. (I have chosen the term *design reabstraction* rather than a recommended term, design recovery,[15] because the latter implies a process that uses more sources of information and results in a broader scope of information about the design.)

The thorniest problem we have with software is its complexity.

Once we have recaptured the conceptual view, we incorporate the enhancement into the conceptual construct. In doing this, we use a different design method than that used for new function. The design of the enhancement involves the *design of a delta* to the existing function. The design must specify how the existing function is to be altered. For many enhancements the design is not isolated design. The new function can be scattered throughout the existing structure.

Approaches to forging a silver bullet

Where should we look for a silver bullet? If we are to forge a silver bullet, it will only be by using those technologies that directly address the essential attributes of software: conceptual content, data and function representation, and multiple subdomains, and the consequences of these attributes. I propose the following four basic approaches, in order of decreasing importance:

1. Support intellectual control over subdomain complexity
2. Provide higher-order constructs
3. Support development of the conceptual construct
4. Support incremental steps during development

Each approach is now described in some detail.

Support intellectual control over subdomain complexity. The thorniest problem we have with software is its complexity. By virtue of the conditions in a program, the number of subdomains increases dramatically as the program increases in size. Developing the conceptual construct for a new system, forward from the requirements to specifications to design to implementation, is relatively easy. When we proceed forward and top-down, we are developing and refining the conceptual construct in a natural way, from the whole to its parts, from overview to details.

However, when we are working with existing code, for example, to develop enhancements, the reverse process is very difficult. The many subdomains of the system present themselves with equal importance; it is difficult to abstract from the code the relevant concepts and recover the conceptual construct for enhancements. It is difficult to maintain intellectual control over the existing complexity. Yet the fact is that 80 percent or more of our development deals with enhancing existing code. Above all we need solutions that will allow us to reabstract the relevant concepts at every level of component hierarchy. We need methods to handle the complexity rigorously, with intellectual control. The technologies I propose for this are design reabstraction and formal verification.

Design reabstraction. Design reabstraction is the process of recovering or rediscovering the higher-level concepts—the design—that motivated an existing segment of code. Generally we cannot rely on whatever existing detailed documentation may exist. The documentation of the lower levels of design is almost invariably out of date because it was not maintained as the software was fixed and enhanced. In contrast, the documentation of the higher levels of design, if it is available, may be useful in establishing the basic conceptual constructs of the system.

The basic approach to reabstraction is to abstract and represent the function of a program fragment from the code itself in a stepwise reabstraction process. Work in design reabstraction has been re-

ported for the Cleanroom methodology[16,17] and the REDO project.[18,19] Both approaches propose a potential automated abstraction system to derive the single-valued functional abstraction of a program fragment. Both approaches attempt to derive a formal, mathematical expression of the function.

In light of the conceptual nature of software, I propose that it will be more fruitful to treat reabstraction as a process of recovering the *conceptual* construct rather than a mathematical representation. Reabstraction is better viewed as the reverse of design, that is, recovering the intermediate concepts that were present when the software was designed. Such a process could never be fully automated because the higher-level conceptual content is absent and needs to be supplied by the developer. However, tools can be developed to provide automated assistance in the process.

The conceptual content at any level can be abstracted from the constructs of the software by identifying the appropriate concepts in a stepwise manner. Choices must be made as to what are the "relevant" details for higher levels, that is, which conditions have conceptual relevance at a higher level and which are merely implementation details. Constructs are thus encapsulated by generalizing concepts, and intellectual control over ever larger segments of the code can be achieved.

The process of reabstraction needs to be tied to the code so that the reabstracted concepts can be traced back to the code that motivated them. The process also needs to be rigorous to the extent that no segment of code can be overlooked.

Formal verification. The proponents of the mathematical treatment of software have long held out the hope that programs can be formally verified or proved. Brooks points out that although formal verification is useful where security or safety are key considerations, it does not promise to save labor. In fact, only a few substantial programs have ever been verified. Moreover, verification can only establish that the program meets its specification, whereas the essential problem is to develop the specification correctly in the first place.

I disagree with this assessment. To begin with, there are methods of verification, perhaps less rigorous than what Brooks had in mind, that can be applied to large systems. For example, the Cleanroom methodology[20] has demonstrated that program verification can be successfully integrated into the development of real-life systems with no loss in productivity and a significant gain in quality. (The Cleanroom methodology includes the methods of box structure specification, function theoretic correctness verification, and statistical usage testing. For the purposes of this discussion, we will consider just the verification portion of the methodology.)

Furthermore, Brooks's view that program verification is no help to us in developing a complete and consistent specification does not take into account the fact that building the specification proceeds from the same thought activity as the development of the program. The process of building the specification *for the purpose of verification* does in fact provide a rigor that would not otherwise be there. Yes, the process of developing the conceptual construct is potentially flawed because we must first form the concepts out of our thinking activity before we can produce either the specification or the implementation. However, any amount of rigor that we introduce into this process will help.

The Cleanroom methodology uses specification of intended function in the form of inputs, outputs, transformation and state data, using mathematical proof arguments to demonstrate the correctness of the implementation or a lower-level design refinement during team reviews. Cleanroom verification provides rigorous methods during formal reviews and encourages the same level of rigor during individual work. The methodology is powerful and provides the rigorous methods needed for intellectual control.

However, in order for Cleanroom verification to be broadly applicable, it will need to include specifications and verification procedures for purely conceptual representations of function. As we saw earlier, the mathematical treatment of programs is limited in the extent to which it can abstract the complex subdomains of the function of the program and still remain mathematical. A mathematical expression of the function of the program will have the equivalent complexity of the code itself. Verification based strictly on mathematically expressed function specifications will necessarily be complex and tedious. I believe

it is to this form of program verification that Brooks alluded.

If we look more broadly at the process of verification, we find that developers ultimately rely on a higher order of verification with their conceptual understanding of the system and the required function, as was mentioned earlier. The determination of correctness thus relies on the thinking ability of the reviewers to be convinced of the correctness, regardless of the form of the notation. Cleanroom proponents appear to be moving in this direction. For example, Pleszkoch, et al.[17] recognize the usefulness of simple human concepts to represent complex functional expressions.

Cleanroom also has restrictions on the use of certain language constructs, for example, multiple exits from a program fragment, loop ITERATE, and loop LEAVE. These appear to be too restrictive, particularly if methods, such as employing partition analysis,[7,8] can be devised that adequately demonstrate the validity of programs containing such constructs. More work in this area is warranted.

The greatest limitation of the Cleanroom methodology, however, is its appropriateness for developing enhancements to existing code. Cleanroom relies on formal specifications of the intended function, but such specifications are invariably absent for existing products not developed using Cleanroom. There are no proven methods for developing the specification of existing code, although the work on design reabstraction (previous section) shows promise. Thus the success of the Cleanroom methodology for most development efforts is dependent on the development of a valid, usable design reabstraction methodology.

Provide higher-order constructs. One attack on subdomain complexity is to prevent the creation of new subdomains by eliminating or hiding the conditional logic that gives rise to them. This requires constructs in the language and the interfaces to other softwave components that will perform higher-order transformations while hiding the conditional details of implementation. Such constructs enable a complete encapsulation of function so that it can be dealt with without regard to the conditions arising in its implementation.

Higher-order constructs also allow the developer to partition the problem into the part that must be developed and the part(s) that already exists and can be reused. Thus the problem to be solved is smaller, and the scope of the development effort is reduced. Such constructs need to be both generic and problem-domain-specific.

The solutions for this basic approach enable us to reduce the number of subdomains with which we must deal.

Generic, higher-order language constructs and reuse. The most useful approach will be to provide higher-order *generic* constructs in several ways:

- Language extensions supported by the compiler—Extensions would include providing built-in functions not originally part of the language (e.g., MAX and MIN functions, spreadsheet @ functions like @VLOOKUP, specialized mathematical or financial functions, etc.). It would also include incorporating built-in abstract data types, for example, for stacks, queues, and maps, with the operations for these data types (e.g., PUSH, POP, QUEUE, DEQUEUE) provided as part of the language. Other language extensions would provide needed built-in transformations that are used regularly in various applications, such as the support of variable nested data structures with self-referencing lengths.

  More broadly, language extensions in our existing languages can support better encapsulation, allow for (better) type checking, and provide for user-defined abstract data types and object-oriented constructs.

- Generic reuse—It would include providing generic constructs in the form of reuse building blocks and object class libraries. Such functions would provide a convenient packaging of complex function with an interface that hides the complexity, for example, reusable parts that provide a simple interface to the graphical user interface function of the operating system.

- Other types of extensions—This would include extensions that provide higher-level function, such as pipelines, compiler macros, and spreadsheet add-in packages. Any facility that can encapsulate the generic function needed will help reduce the complexity of the program. For example, the following VM/CMS (Virtual Machine/

Conversational Monitor System) pipeline[21] construct reads the records in a file, reformats them, sorts the reformatted records, eliminates duplicates, and stores the results in a stem (array) variable. This single line of code is equivalent to about 25 lines of a language like REXX or PL/I:

```
PIPE < file | spec 73-80 1 | sort | unique
            | stem tbl.
```

There is no logical limit to the level of higher-order constructs that can be provided. We can develop still-higher-order constructs from the lower ones. However, Brooks feels that there is a limit to the usefulness of higher-order constructs. They will ultimately create a "tools mastery problem that increases, not reduces the intellectual task of the user who rarely uses the esoteric constructs."[1] Brooks may be right, but we are far from that limit in most high-level languages. Rather than discourage the creation of higher-level constructs, I would propose aggressive selection and development of such constructs. Indeed, it should be the responsibility of language standards bodies and compiler developers to seek out and provide such extensions as part of their language. The language should never be viewed as complete and fixed. Developers also have a concomitant responsibility to be aware of new language features and to use them.

The selection of functions for generic reuse requires that the problem domain concepts, function, and data representation all be suitable for general use. Most existing code in an application is not suitable for general reuse because of mismatches in one or more of these areas. In particular, the problem domain concepts or the data representation, or both, may not be general enough. Finding reusable parts is difficult.[22]

The use of higher-order constructs is applicable to developing enhancements to existing code products because the new constructs can be introduced as enhancements are added. Indeed, developers should be constantly seeking ways to upgrade their product by introducing such features as part of an overall upgrade plan.

Problem-domain-specific language constructs and reuse. In addition to generic language constructs and reuse, it is also possible to develop and use *problem-domain-specific* constructs, that is, higher-order constructs for a specific problem domain. This would include the use of reusable parts or object class libraries for a specific problem area, for example, reusable payroll tax calculation modules, class libraries of functions to support satellite orbital calculations, or geological models for oil exploration.

Reuse works in problem-domain-specific applications despite possible mismatches in problem domain concepts or data representation.

Reuse works in problem-domain-specific applications despite possible mismatches in problem domain concepts or data representation. The reusing application can make the rest of the application conform to the requirements and capabilities of the reused parts.

Higher-level languages. CASE (computer-aided software engineering) tool vendors hold out great hope for generating code directly from the design representations in their tools. However, if such a capability is to be available, their designs will need to provide all of the details that currently go into the code. The conditions that must be tested and the transformations that must be performed, in short, the entire set of subdomains that would ordinarily be produced in the code, will need to be specified in their designs. The complexity of multiple subdomains is an essential attribute, and inevitably code generators will need to be able to specify all subdomains.

In order to specify such a level of detail, the CASE tool must provide constructs that specify transformations, conditional branching (IF-THEN-ELSE), and repetitive operations (loops), in other words, all of the basic features of a programming language. Thus, code generators will be nothing other than higher-level languages. If we are fortunate, they will also provide various higher-order constructs (see previous section).

Unfortunately, such tools will not be readily usable for existing code products because it will be necessary to reabstract the existing function into the design representation of the tool in order for it to be generated again by the tool. For most applications, such reabstraction will not be practical.

Brooks notes the recent widespread use of generalized tools such as databases, spreadsheets, graphics packages, and statistical packages. These tools enable nonprogrammers to develop their own applications via simple programming (e.g., via macros, scripts, program generator specifications, etc.). These tools in effect provide higher-order constructs and the appropriate execution environment (spreadsheet, database, graphics) so that the nonprogrammer user can develop specialized applications without much technical expertise. Yet what is being done is still a form of programming. These languages usually still provide the standard sequential, conditional, and looping constructs that are present in programming languages. The "developer" must still develop the conceptual construct, and the resulting "program" is still subject to all of the essential characteristics of software: complexity, conformity, etc. This approach works because the applications tend to be small and specialized such that their developers can manage them.

Support development of the conceptual construct. Brooks holds that the hard part of building software is the specification, design, and testing of its conceptual construct, not the labor of representing it and testing the fidelity of the representation. The conceptual components of the task of software development now take most of the time. Thus, we should consider those attacks that address the formulation of the conceptual structure. Tools and methods are needed that readily enable development of specifications and designs, permitting clarity of thinking and ease of communicating the conceptual view.

Again, it is much easier to develop the conceptual construct the first time, in the first release of the product. Yet the fact is that 80 percent or more of our designs are enhancements to an existing design. Thus our tools and methods need to assist the design of enhancements to existing code. We must first understand the existing function, formulate the conceptual construct of the enhancement, and then express the design as an incremental enhancement on the base function. Our tools and methods must help formulate the conceptual construct of the enhancement.

The solutions for this basic approach enable us to address the concepts and representation of the conceptual construct, allowing easy development of specifications and designs and permitting clear thinking and communication of the conceptual constructs. Our tools and methods above all need to assist the design of enhancements to existing code.

The technologies that address this area are: program understanding tools, incremental design tools, object-oriented technology, great designers, design execution, and buy versus build.

Program understanding tools. Program understanding is an additional step in the process of design that is absent from new code development. However, for enhancements to existing code, it is critical that the developer investigate and understand how the existing structure works before beginning the design of any enhancements. Program understanding function might ideally include navigation from high-level design representation through module call structures to the code itself, automatic identification of dependencies and interfaces, and automatic creation of data dictionary entries from existing structures.

Program understanding tools are intended to assist the developer in formulating a complete conceptual view of the existing software. Ideally, their function is based on what information the developer requires and in what form. These items are in turn dictated by the internal thought processes the developer uses. Program understanding tools should provide the relevant information while hiding all other details. See References 13 and 23 for excellent summaries of this field.

Program understanding tools are under development within IBM.[24,25] New approaches to the presentation of program information are also being studied inside and outside IBM, for example, program slicing.[26]

Design tools for incremental enhancements. CASE design tools generally do not support designs of incremental enhancements to existing function,[27,28] despite the fact that 80 percent or more of development is done on existing code

products. What is needed is the ability to represent the design of the existing function and the design of the new function (the *design delta*) with highlighting.

A methodology incorporating incremental design would provide program understanding and design reabstraction in an appropriate design representation (see previous sections), redocumentation to capture ideas uncovered during program understanding and other investigation, and the specification and highlighting of the design delta within the base design.

Object-oriented technology. Object-oriented technologies, that is, object-oriented analysis, design, and programming, define data in terms of object types, or *classes*. A class defines a logical collection of data with its associated operations, or *methods*. For example, a Window in a graphical user interface may be a class with a number of methods that maximize, minimize, and move the window on the screen. More restrictive classes may be defined as *subclasses* of a class. The subclass *inherits* the methods of its superclass but may have additional methods or slight variations on the methods of its superclass. For example, a ScrollableWindow might be a subclass of the class Window, with additional methods to display a scrolling bar and perform scrolling in the window. It would inherit the methods from its Window superclass such as maximizing and moving but may have slightly different logic for its minimizing method.

Specific instances of a class are called *objects*. Thus, a user might cause two Window objects and three ScrollableWindow objects to be created during a particular session at a terminal. Some object-oriented languages support *multiple inheritance* whereby a class may inherit methods from multiple different superclasses. For example, the ScrollableWindow class might inherit a MouseButtonHold method from a more general Mouse class to support scrolling with the mouse button held down.

The overall function of an application can be completely encompassed by classes defined for the application and their methods. Alternatively, there may be functionally oriented segments of code that use the classes and methods to accomplish the overall tasks of the application.

Brooks feels that object-oriented programming is a promising new technology: "Many students of the art hold out more hope for object-oriented programming than for any of the other technical fads of the day. I am among them."[1] However, in Brooks's view object-oriented programming addresses only accidental aspects of development: it allows the developer to express the essence of the design without having to express large amounts of syntax and allows a higher-order expression of the design. But the complexity of the design is the essence, not its expression. Thus, Brooks does not expect an order of magnitude improvement because type specification is not nine-tenths of the work of designing a software entity.

Object-oriented technologies define data in terms of object types, or classes.

I disagree in part with Brooks's assessment. Brooks overlooks the fact that object-oriented technologies permit a completely different conceptual approach to the conceptual construct (structuring via its data rather than its function) and therefore do address the essence of the problem: they provide a framework for a different, possibly more powerful expression of the design.

More specifically, object-oriented technologies assist in two of the essential aspects of software: the conceptual content and representation. At least in some problem domains the data orientation has the advantage of a more natural representation. I would expect designs based on data representations to exhibit more stability with change over time, greater ease in maintaining the conceptual integrity of the design, and better overall encapsulation. Moreover, object-oriented technologies assist reuse (see previous sections). Again, their naturalness of representation helps with problem-domain-specific reuse. I would expect class libraries to be easier to generalize for reuse than function-oriented code.

Initial reports from studies of the use of object-oriented technologies support these expectations. In three studies, object-oriented programming showed promise in increasing productivity because of the ease of reuse and ease of maintenance.[29–31] The ease of reuse was demonstrated by the amount of class code reused by inheritance. The ease of maintenance was demonstrated in terms of less effort to implement a change, fewer lines needing to be added or changed, more localized changes, and fewer interface changes for enhancements and extensions to function. But the evidence also shows that object-oriented technologies are not a silver bullet: there was no tenfold improvement in quality or productivity.

Moreover the use of object-oriented technologies is not without a price. A number of issues are raised in the literature:

- Inhibits comprehension of overall function—The calling structure of object-oriented applications tends to be deeper and more finely differentiated (deeper hierarchies with larger fanout) than functional implementations. This condition leads to a separation and dispersion of function across classes. Wilde and Huitt[32] confirm this result and add that the hierarchical class structure multiplies the kinds of relationships that must be considered when changing an object-oriented program. Although the separation of function increases the possibility for reuse, it also increases the difficulty of tracing function and actually inhibits comprehension.

- Hidden or unforeseen dependencies—In discussing requirements for object-oriented testing, Perry and Kaiser[33] point out that the property of inheritance tends to make the effects of changes more difficult to determine and understand because of dependencies. These dependencies may be caused by complex inheritance structures, methods that use other methods at the same or higher levels, and multiple inheritance. For example, when we modify an existing subclass, there may be hidden dependencies in the methods inherited from a superclass. A method may use different subordinate methods from different classes, depending on the class inheritance structure, causing hidden dependencies. In addition, late binding of methods based on type also can cause hidden dependencies.[32]

  All of these effects result in additional unforeseen, unobvious dependencies that can cause more complex rather than less complex and manageable designs. Rather than simplifying and removing accidental difficulties, the object-oriented paradigm appears to add accidental difficulties.

- Need for object-oriented process and tools—Goyal[34] argues for a development process specifically tailored to object-oriented design that would include a domain analysis step in which objects are defined, developed, and refined, as well as the more conventional function analysis and design steps. Also needed are supporting tools such as for class browsing, cross-referencing, configuration management, and change control.[32,35]

- Difficult to learn to use—The claim from proponents that object-oriented design is natural and therefore easy to learn was refuted by at least one researcher[36] who found that programmers experienced in functional programming had difficulty decomposing a problem, identifying classes and methods, and generally implementing a solution in an object-oriented language.

With object-oriented development, as with function-oriented development, it is necessary to be able to handle multiple simultaneous releases, large releases, and large teams. This type of development implies that tools are needed to coordinate the class libraries across diverse projects and teams.

Most importantly, techniques must be developed to introduce object-oriented constructs into existing functionally oriented applications. To accomplish this task, object-oriented extensions have to be developed and supported in the existing programming languages, and techniques have to be developed for gradually converting existing data structures and function into object classes and methods. Some promising work in this area has begun.[37]

Until object-oriented technologies become more widely understood and used, and until a large set of products have been developed with them, acceptance of this technology will be inhibited. Developers working on function-oriented products will continue to do so, whereas developers work-

ing on object-oriented projects will use their own development methods. Until techniques are available to allow object-oriented technology to be utilized in existing products, the two orientations will continue on separate paths with little cross-fertilization or movement of expertise.

Great designers. Brooks proposes that a great designer produces exemplary designs primarily because of inherent talent and this is something that cannot be taught but only cultivated in the talented designers that we may have. Therefore, we should use strategies to identify and cultivate our great designers so as to obtain the most from their talent.

Brooks may be right, but I would suggest that a deeper understanding of the creative thought process and the essential nature of software may make it easier to identify the faculties and abilities that great designers seem to have. Although many of the great designer's thought processes and guiding principles are probably unspoken and some are possibly ineffable, it should not be assumed that we cannot understand more about how a great designer works. With such an understanding it may be possible to teach these principles and develop them in others.

For example, it is clear that a great designer must have extreme facility in handling the conceptual construct of a system and a good sense of the conceptual principles that went into the design. How the great designer achieves this should be of use to us all.

Design execution. In response to Brooks[1] and Parnas,[4] David Harel[2] offers design execution as an approach that will assist the developer in prototyping, simulating, and testing the design representation before it is committed to implementation. Design execution can cause unforeseen situations to surface resulting from weaknesses, contradictions, and flaws in the conceptual structure (what I termed earlier "major conceptual errors").

This approach is useful because it allows the developer to work with the conceptual construct of the system while it is being developed, to clarify the thinking about the system, and to uncover unforeseen situations and interactions between the different parts of the system.

However, this method does not handle the enhancement of existing code well. It requires us to reabstract the existing design in order to take advantage of design execution for enhancements to the existing design. In some cases the new function will be sufficiently isolated from the existing function, and in some cases the existing design can be readily reabstracted, but in many cases this will not be possible. But for new function, and cases where we can reabstract easily, design execution provides a powerful tool for developing the design.

Iterative development needs an overall structure and development plan.

Buy versus build. Brooks proposes the use of off-the-shelf products to meet the needs of the organization, rather than specialized, custom-built software systems. He proposes use of specialized packages, for example, for payroll, inventory control, accounting, etc. Many such products are available today at reasonable prices.

Specialized packages eliminate the need to develop custom software. The conceptual construct has already been developed and implemented. If there are instances where the package does not exactly conform to the needs of the organization (for example, the payroll system does not accommodate different pay rates for the same employee for different types of work), the organization conforms to the capabilities of the package and adapts its practices to it. This is a dramatic reversal of requiring software to conform to human institutions, but the cost benefit to the organization to use a package rather than build a custom system frequently justifies it.

The use of specialized packages is valid for many software applications but does not address the more common problem of applications for which there are no packages or the problem where we already have existing software that must be maintained and enhanced. For these cases, the more generalized approach of reuse of software parts or

building blocks is a valid extension of Brooks's idea.

Support incremental steps during development. Software development is a thinking activity, yet our thinking faculty has limits both in breadth and through time. We do not generally develop the concepts of the system in all their clarity and detail all at once, nor can we hold them all in mind at one time. We must work to develop the conceptual construct and its representations over time. We must consult others' viewpoints and reactions and then repeat and refine our thinking through a number of iterations. We must explore the implications of our thinking thus far in various directions, and again refine what we have developed in further iterations. This process of iterative thought refinement proceeds from requirements definition through implementation.

We cannot do this development process all at once for any but the smallest problems. For most development projects, however, large efforts are a fact of life. Thus, we need methods and tools that allow us to partition the problem into smaller pieces and iterate on each step of the process until the refinement is complete at that level.

The solutions for this basic approach are those that enable us to develop and refine our thinking over time, and to partition the problem into smaller, more manageable chunks. This enables us to make use of the principle of divide and conquer. We accomplish these ends by iterative development and refinement. The notion of iterative development and refinement apply in two places: requirements and design.

Requirements refinement and prototyping. Brooks asserts that the hardest single part of development is deciding what to build. To make this decision he recommends that we develop the requirements and external specifications of the system by iteratively extracting and refining them with the client or customer. Without such a procedure it is practically impossible to specify completely, precisely, and correctly the exact requirements. Prototyping can be used to simulate the externals and important interfaces and to present the main functions of the intended system to users for validation.

Incremental (iterative) development. Again, Brooks proposes that a better analogy for software development is the gradual growth of the system rather than the construction and building of the system. We should grow the software construct gradually rather than build the entire large structure in one step. This process allows a kind of prototyping to occur as we grow each increment of the system and add it to the base. The system evolves or grows gradually and we are able to take stock of what we have developed so far and make adjustments where needed. An example of the use of this method in IBM is the Experimental Software Development Center.[38]

Iterative development needs an overall structure and development plan: a high-level design and an externals specification at least to some level of detail. Also needed is a plan for partitioning the function into increments with demonstrable function that can be tested as a unit. Then each team—usually they are small teams—proceeds with development and test of its increment. When the increment has been tested to a level of confidence, it is added to the base product. Proponents of iterative development claim that code can be developed in this way with high productivity and good quality.

One pitfall of this method is the potential to lose sight of the fact that we must go through the required steps of development, that is, through the necessary intellectual steps and through the different levels of detail in the definition of the product structure. If we ignore these steps, we end up doing them informally, without the proper communication of critical design decisions, etc.

Recommendations

Does a silver bullet exist in any of these technologies? At this point we cannot tell. However, we can say that the most difficult essential attribute of software is its multiple subdomains. And the most difficult consequence is the need to maintain and enhance existing code. A method that directly attacks both of these areas has a chance.

When we develop new function we go through elaborate steps to develop the conceptual construct, to ever greater levels of detail. We develop the code, and then we gradually let go of our conceptual view. What was once vividly alive in our thinking fades. Formal documentation of the system becomes out of date. Low-level or module-level design documentation is generally dis-

carded, for example, when the code is completed. We are left with incredibly detailed, complex code and not much else that reliably presents or explains its conceptual elements. The higher levels of documentation may be lost or become so out of date that they are no longer trustworthy. Over time documentation in the code itself, even the comments at the statement level, can become suspect as the code is modified again and again.

Thus, we are left with a gap between any accurate conceptual construct and the code. There are no longer any accurate intervening details of the design between whatever high-level concepts we may have and the code. When we attempt to tackle any problem relative to the system—be it developing fixes or adding new function to the code, we are faced with the complexity of the code and precious little else. It is a gap that in practice is impossible to close with accuracy for any but the simplest programs. Our minds simply can no longer encompass all of the code. We are forced to rely on imperfect, error-prone methods. Our mental prowess and that of our colleagues are unequal to the task. We make mistakes that we cannot help.

But if the source of this problem is the combination of the essential attributes of software: conceptual content, representation, and multiple subdomains, and the fact that we must deal with the existing code, the solution lies in precisely those areas: the reabstraction of the conceptual construct into higher-order *conceptual* abstractions that encapsulate the myriad subdomains. The reabstraction process reverses the process of design. This approach seems to me to be the only one that will bridge the gap facing almost all developers, between the code and a reliable representation of its conceptual construct.

Thus, my recommendations toward forging a silver bullet for software development technology are:

1. *Develop and deploy technologies for design reabstraction and program understanding*—Design reabstraction is the technology that deserves the greatest focus, in my opinion. Not only does it have significant promise in the most difficult essential aspects of software, but it is also the key element or a significant dependency for a number of other promising technologies if they are ever successfully to deal with existing code: formal verification, object-oriented technology, design execution, and design tools for incremental enhancements. Once adequate reabstraction technology is developed, these other technologies become much more viable for legacy systems.

   Likewise, program understanding technology is a key element in enabling developers to formulate the conceptual view of the system easily and accurately. In many ways, program understanding technology is a prerequisite to design reabstraction because the developer must first understand the existing function in order to reabstract it.

2. *Develop and deploy those technologies that can readily be introduced into legacy system processes*—Several other technologies offer the advantage of being readily introducible into our legacy system processes, that is, without needing to re-engineer or reabstract the existing logic. These technologies include higher-order language constructs, generic and problem-domain-specific reuse, and incremental (iterative) development. With a plan for software revitalization and process improvement, such technologies can be introduced gradually as the system is enhanced.

Conclusion

To repeat the basic proposal of this paper: if a silver bullet is possible, it can only be forged by directly addressing the essence of software. Even if we are not able to achieve an order of magnitude improvement in quality and productivity, these approaches will constitute the most profitable and possibly the only effective attack on the problem.

I believe a radical shift in our approach to software development technology is warranted. We need to recognize that software development is above all else a human endeavor, one that challenges our thinking faculties to the limit. The focus of our technologies must be on the central role of the developer who uses intense thinking activity as an integral part of the process. In using our technologies we must realize that it is the human understanding of the concepts of the problem that is important and that all the processes, methods, notations, and syntax are really only secondary to that understanding.

Acknowledgments

I am very grateful for the helpful comments I received on earlier drafts of this paper from Yoshihiro Akiyama, Mike Schaul, Ben Sun, Harry Cook, Lloyd Gilliam, Chuck Brabec, Roy Klaskin, and the referees.

Appendix: Estimating the subdomains of a program

Every program fragment can be partitioned into a number of subdomains from its conditional statements, such as IF-THEN-ELSE, SELECT-END, and DO WHILE-END. Different statements *split* the subdomain of a program in varying amounts. Statements with no conditions do not split the subdomain. The IF-THEN, IF-THEN-ELSE, and DO WHILE-END statements split the subdomain in two. The SELECT-END statement splits the subdomain by the number of WHEN and OTHERWISE clauses. We can estimate the lower and upper boundaries of the number of subdomains from the number of conditional statements.

The conditional statements interact in two ways, sequentially and nested, to combine subdomains. Sequential statements form a cross product of the subdomains of each statement to produce the subdomains of the combined set of statements. For example, two sequential IF-THEN-ELSE statements each have two subdomains. Together, they have a total of four subdomains formed by the cross product of the subdomains of each of them:

```
if condition-A then transform-W
else transform-X

if condition-B then transform-Y
else transform-Z
```

Subdomains:

```
1. condition-A
   #AND# condition-B
      → transform-W, transform-Y

2. condition-A
   #NOT# condition-B
      → transform-W, transform-Z

3. #NOT# condition-A
   #AND# condition-B
      → transform-X, transform-Y

4. #NOT# condition-A
   #NOT# condition-B
      → transform-X, transform-Z
```

Nested statements form a cross product of their subdomains with the condition of the statement in which they are nested. For example, an IF-THEN-ELSE nested within one leg of another IF-THEN-ELSE forms a cross product with the outer IF-THEN condition and produces a total of three subdomains:

```
if condition-A then do
   if condition-B then transform-X
   else transform-Y
end
else transform-Z
```

Subdomains:

```
1. condition-A
   #AND# condition-B
      → transform-X

2. condition-A
   #NOT# condition-B
      → transform-Y

3. #NOT# condition-A
      → transform-Z
```

A program fragment has any number of combinations of sequential and nested conditional statements. Thus the total number of subdomains falls between the total for nested combinations and sequential combinations.

For the sequential combinations, all of the subdomains for all prior statements are split as we add each new conditional statement. If we take as an example a 10 000-statement program containing 2000 conditional statements occurring sequentially, and we assume that each conditional statement splits the subdomain in two, as we add the next conditional statement we have the relationship shown in Table 1.

For combinations of nested conditional statements, as we add each new conditional statement we nest it within a higher-level conditional statement. Thus at each nesting level, we can add as many new conditional statements as there are places available to put a nested statement. The resulting structure is a completely filled binary tree. The number of subdomains is the number of

Table 1 Subdomains for sequential conditional statements

Added Conditional Statements	Total Conditional Statements	Total Subdomains
1	1	$2 = 2^1$
1	2	$4 = 2^2$
1	3	$8 = 2^3$
1	4	$16 = 2^4$
⋮	⋮	⋮
1	2000	2^{2000}
⋮	⋮	⋮
—	n	2^n

Table 2 Subdomains for nested conditional statements

Added Conditional Statements	Total Conditional Statements	Total Subdomains
1	$1 = 2^1 - 1$	$2 = 2^1$
2	$3 = 2^2 - 1$	$4 = 2^2$
4	$7 = 2^3 - 1$	$8 = 2^3$
8	$15 = 2^4 - 1$	$16 = 2^4$
⋮	⋮	⋮
1024	$2047 = 2^{11} - 1$	$2048 = 2^{11}$
⋮	⋮	⋮
—	n	n + 1

leaves on the tree. We can represent this as shown in Table 2.

The total subdomains for nested conditional statements grow as n + 1, whereas the subdomains for sequential conditional statements grow as 2^n. The total subdomains for a program fragment lies between these two bounds. Thus, for the 10 000-statement program with 2000 conditional statements, the total subdomains would be somewhere between 2001 and 2^{2000}.

Cited references and notes

1. F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer* **20**, No. 4, 10–19 (April 1987).
2. D. Harel, "Biting the Silver Bullet: Toward a Brighter Future for System Development," *Computer* **25**, No. 1, 8–20 (January 1992).
3. B. Johnson, "Is the Silver Bullet Knowledge-Based?" *IEEE Software* **10**, No. 1, 10 (January 1993).
4. D. L. Parnas, "Software Aspects of Strategic Defense Systems," *Communications of the ACM* **28**, No. 12, 1326–1335 (December 1985).
5. I am using the term "forge" in the sense of forming or shaping a metal object by heating and hammering. Forging a metal bullet is an apt metaphor for developing what is needed to overcome the essential complexity of software.
6. Brooks uses the term "representation" more restrictively to mean the representation of the conceptual construct—the design—in a programming language. I am using it here more broadly to mean the representation of the concepts in the design as well as the code.
7. D. J. Richardson and L. A. Clarke, "Partition Analysis: A Method Combining Testing and Verification," *IEEE Transactions on Software Engineering* **SE-11**, No. 12, 1477–1490 (December 1985).
8. D. J. Richardson, *A Partition Analysis Method to Demonstrate Program Reliability*, Ph.D. dissertation, University of Massachusetts, Amherst, MA (September 1981).
9. The discussion of partitioning a program into its subdomains is necessarily limited here. The subdomains for all executable language constructs can be identified. However, some constructs are problematic. For example, looping constructs can be treated as repetitive occurrences of their conditions and transformations. The resulting subdomains may contain indeterminate repetition factors. The GOTO construct itself does not create additional subdomains but is an indication that some other conditional construct was used.
10. R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley Publishing Co., Reading, MA (1979).
11. H. D. Mills, V. R. Basili, J. D. Gannon, and R. G. Hamlet, *Principles of Computer Programming: A Mathematical Approach*, Allyn and Bacon, Boston, MA (1987).
12. R. L. Glass, "Creativity and Software Design: The Missing Link," *Information Systems Management* **9**, No. 3, 38–43 (Summer 1992).
13. T. A. Corbi, "Program Understanding: Challenge for the 1990s," *IBM Systems Journal* **28**, No. 2, 294–306 (1989).
14. A. Lakhotia, "Understanding Someone Else's Code: Analysis of Experiences," *Reverse Engineering Newsletter* No. 4, 6–8, in *Software Engineering Technical Committee Newsletter* **11**, No. 3 (January 1993).
15. E. J. Chikofsky and J. H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software* **7**, No. 1, 13–17 (January 1990).
16. P. A. Hausler, M. G. Pleszkoch, R. C. Linger, and A. R. Hevner, "Using Function Abstraction to Understand Program Behavior," *IEEE Software* **7**, No. 1, 55–63 (January 1990).
17. M. G. Pleszkoch, P. A. Hausler, A. R. Hevner, and R. C. Linger, "Function Theoretic Principles of Program Understanding," *Proceedings of the 23rd Annual Hawaii International Conference on System Sciences* **2**, 74–81 (1990).
18. P. T. Breuer and K. Lano, "Creating Specifications from Code: Reverse-Engineering Techniques," *Software Maintenance: Research and Practice* **3**, No. 3, 145–162 (September 1991).
19. K. Lano, P. T. Breuer, and H. Haughton, "Reverse-Engineering COBOL via Formal Methods," *Software Maintenance: Research and Practice* **5**, No. 1, 13–35 (March 1993).
20. H. D. Mills, M. D. Dyer, and R. C. Linger, "Cleanroom Software Development," *IEEE Software* **4**, No. 5, 19–25 (September 1987).
21. *Virtual Machine/Enterprise Systems Architecture CMS Pipelines Reference, Release 1.1*, SC24-5592-00, IBM Corporation (February 1992); available through IBM branch offices.

22. K. Sikkel and J. C. van Vliet, "Perspectives of Software Re-usability," *Software Re-use, Utrecht 1989: Proceedings of the Software Re-use Workshop*, Springer-Verlag, London (1989), pp. 131–136.
23. D. J. Robson, K. H. Bennett, B. J. Cornelius, and M. Munro, "Approaches to Program Comprehension," *Journal of Systems and Software* **14**, No. 2, 79–84 (February 1991).
24. P. Brown, "Integrated Hypertext and Program Understanding Tools," *IBM Systems Journal* **30**, No. 3, 363–392 (1991).
25. L. Cleveland, "A Program Understanding Support Environment," *IBM Systems Journal* **28**, No. 2, 324–344 (1989).
26. K. B. Gallagher and J. R. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Transactions on Software Engineering* **17**, No. 8, 751–761 (August 1991).
27. R. G. Mays, "CASE Requirements for Enhancing Existing Systems," *Fourth International Workshop on Computer-Aided Software Engineering (CASE '90)*, IEEE Computer Society Press, Los Alamitos, CA (December 1990), pp. 16–17.
28. E. Bush, "CASE for Existing Systems," *Information Strategy: the Executive's Journal* 7, No. 3, 31–39 (Spring 1991).
29. D. Mancl and W. Havanas, "A Study of the Impact of C++ on Software Maintenance," *Proceedings, Conference on Software Maintenance, San Diego, CA, 1990*, IEEE Computer Society Press, Los Alamitos, CA (1990), pp. 63–69.
30. S. Henry, M. Humphrey, and J. Lewis, "Evaluation of the Maintainability of Object-Oriented Software," *IEEE Region 10 Conference on Computer and Communication Systems, Hong Kong, September 1990* (1990), pp. 404–409.
31. J. A. Lewis, S. M. Henry, D. G. Kafura, and R. S. Schulman, "On the Relationship Between the Object-Oriented Paradigm and Software Reuse: An Empirical Investigation," *Journal of Object Oriented Programming* **5**, No. 4, 35–41 (July/August 1992).
32. N. Wilde and R. Huitt, "Maintenance Support for Object Oriented Programs," *Proceedings, Conference on Software Maintenance, Sorrento, Italy, October 1991*, IEEE Computer Society Press, Los Alamitos, CA (1991), pp. 162–170.
33. D. E. Perry and G. E. Kaiser, "Adequate Testing and Object-Oriented Programming," *Journal of Object Oriented Programming* **2**, No. 5, 13–19 (January/February 1990).
34. P. Goyal, "Issues in the Adoption of Object-Oriented Paradigm," *COMPCON Spring '91, Digest of Papers*, IEEE Computer Society Press, Los Alamitos, CA (1991), pp. 534–539.
35. M. Letjer, S. Meyers, and S. P. Reiss, "Support for Maintaining Object-Oriented Programs," *Proceedings, Conference on Software Maintenance, Sorrento, Italy, October 1991*, IEEE Computer Society Press, Los Alamitos, CA (1991), pp. 171–178.
36. F. Détienne, "Difficulties in Designing with an Object-Oriented Language: An Empirical Study," *Proceedings of the IFIP TC 13 Third International Conference, August 1990 in Human-Computer Interaction—INTERACT '90*, Elsevier Science Publishers B.V., North-Holland (1990), pp. 971–976.
37. C. L. Ong and W. T. Tsai, "Class and Object Extraction from Imperative Code," *Journal of Object Oriented Programming* **6**, No. 1, 58–68 (March/April 1993).
38. J. J. McInerney and T. A. Corbi, *IBM's ESDC: People, Process and Productivity*, IBM Research Report, RC 16692 (#73967), IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (March 1991).

Accepted for publication November 4, 1993.

Robert G. Mays *IBM Networking Software Division, RTP Networking Laboratory, P.O. Box 12195, Research Triangle Park, North Carolina 27709 (electronic mail: mays@ralvm12.vnet.ibm.com).* Mr. Mays is a senior programmer in the Software Process, Assessments, and Technology department. He worked on management systems applications, ranging from minicomputer process control systems to mainframe database/data communications systems, at Eastman Kodak Company for 12 years prior to joining IBM in 1981. Mr. Mays is currently involved in developing and deploying new software technologies, including improved methods for enhancing existing software products using program understanding and design reabstraction. He is also involved in the enhancement and deployment of the Defect Prevention Process throughout IBM. He was one of the recipients of IBM's first Corporate Quality Award in recognition for the development of the Defect Prevention Process. Mr. Mays received his B.S. degree in chemistry in 1968 from the Massachusetts Institute of Technology and is a member of the Association for Computing Machinery, the IEEE Computer Society, the Software Maintenance Association, and the American Society for Quality Control.

Reprint Order No. G321-5531.