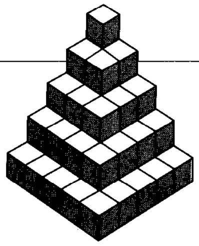

Recognizing Design Decisions in Programs

Spencer Rugaber*, *Stephen B. Ornburn*, and *Richard J. LeBlanc, Jr.
Georgia Institute of Technology

Recognizing design decisions is essential to maintaining and reverse-engineering code. But you first need a way to characterize these decisions and their underlying rationales.

Maintenance, reverse engineering, and reuse rely on being able to recognize, comprehend, and manipulate design decisions in source code. But what is a design decision? We have derived a characterization of design decisions based on the analysis of programming constructs. The characterization underlies a framework for documenting and manipulating design information to facilitate maintenance and reuse activities.

During program development, many decisions are made. Some address the problem domain and how it should be viewed and modeled. Others address constraints imposed by the solution space, including the target machine and language. Some decisions stand alone and have little effect on the rest of the program. Others are subtly interdependent. Sometimes decisions are explicitly documented along with their rationales, but more often the only indication of a decision is its resulting influence on the source code.

To effectively maintain an existing system, a maintenance programmer must be able to sustain decisions made earlier unless the reasons for the decisions have also changed. To accomplish this, he must recognize and understand the decisions.

Design process. Software design is the process of taking a functional specification and a set of nonfunctional constraints and producing a description of an implementation from which source code can be developed.

Whether formal or informal, functional specifications are primarily concerned with what the target system is supposed to do, not with how it is to do it. Source code is inherently formal. Although its primary purpose is to express solutions to problems, other concerns like target-machine characteristics intrude. The middle ground between specifications and code is more nebulous. Dallas Webster has sur-

0740-7459/90/0100/0046/$01.00 © 1990 IEEE

veyed the variety of notations and graphical representations used to handle this middle ground.[1]

You can describe the design process as a whole as repeatedly taking a description of intended behavior (whether specification, intermediate representation, or code) and refining it. Each refinement reflects an explicit design decision. Each limits the solution to a class of implementations within the universe of possibilities.

Rationale missing. Design involves making choices among alternatives. Too often, however, the alternatives considered and the rationale for making the final choice are lost.

One reason design information is lost is that commonly used design representations are not expressive enough. While they are adequate for describing the cumulative results of a set of decisions, particularly about the structure of components and how they interact, they do not try to represent the incremental changes that come with individual design decisions.

They also fail to describe the process by which decisions are reached, including the relevant problem requirements and the relative merits of the alternative choices.

The well-known tendency for system structure to deteriorate over time is accelerated when the original structure and design intent are not retained with the code.

Design decisions are not made in isolation. Often, a solution is best expressed through several interrelated decisions. Unless the interdependencies are explicitly documented, the unwary maintenance programmer will fail to notice all the implications of a proposed change. For example, if several pieces of source are interdependent and one piece is changed during maintenance, the others must be changed to sustain the design dependency and to preserve the program's correct functioning. Elliot Soloway and his colleagues have called these situations "delocalized plans" and described specific instances that are difficult for programmers to handle.[2]

Benefits of capturing rationale. If design decisions and their rationale were captured during initial program development, and if a suitable notational mechanism existed to describe their interdependencies, then several aspects of software engineering would profit:

- Initial development would benefit from the increased discipline and facilitated communication provided by the notation.
- Opportunities for reuse would be multiplied by the availability of design information that could be reused as is or transformed to meet new requirements.
- Maintenance would be vastly improved by the explicit recognition of dependencies and the availability of rationale.

Characterizing decisions

In studying various areas, computer science has revealed several categories of design decisions. Abstraction mechanisms

Languages and design decisions

There is a correspondence between the categories of design decisions and the variety of approaches to language design found in modern programming languages. This is not accidental but reflects the fact that languages are designed to make program development easier. They do this by providing a variety of abstraction mechanisms:

- Algol-60, Algol-68, and Pascal introduced and systematized control and data structures. They provided mechanisms to support decomposition of procedures into statements and data into its components. Of course, procedural abstraction has been with us since the early days of languages in the form of procedures and functions.
- Variables, too, have been part of programming since its beginning, but the explicit trade-offs between data and procedures have become more prominent with the advent of functional languages. Programming in a functional language is difficult for a traditional programmer accustomed to using variables.
- Similarly, logic languages like Prolog highlight the function/relation dichotomy. The same kind of conceptual barriers confront a new Prolog programmer used to more traditional styles of programming or even used to a functional style.
- The issue of encapsulation is explicitly stressed in languages like Ada, Modula, and Clu. The programmer expresses the functional interface to a module in a separate construct from the implementation. The idea is to insulate the rest of the code from subsequent maintenance activities that alter the module's implementation while leaving the functional interface unchanged.
- Generalization/specialization is a primary consideration in Smalltalk. The class hierarchy expresses how subclasses specialize their parents. Dynamic binding invisibly delegates computation responsibility to the least general class able to handle it. Ada supports compile-time specialization of generic packages and procedures by data type and functional parameters.
- Representation is supported in most languages, but Clu emphasizes the distinction between representation and specialization by providing separate language constructs for expressing them. Gypsy has features to describe both ideal behavior and implementation details. It supports the proof of their equivalence via semiautomated means.

Encapsulation in Ada

The Ada package construct was designed to support encapsulation via information hiding. You achieve information hiding by separating the description of the facilities that a module provides (its functional specification) from the description of how it provides them (its body or implementation). But even with such a well-designed construct, there remains a range of ways to use encapsulation in Ada.

Imagine an application that requires lines of text to be read from an input file and broken up into words. The words will be statistically analyzed based on properties like their length and position. It is natural to think of a word as being a candidate for encapsulation within a package. This is not just a binary choice; there are a variety of ways that you might address the issue in Ada:

• No encapsulation at all. Read, parse, and analyze the words in the context of the controlling (client) code.

• Hide the reading and parsing in a subprogram (Get_Next_Word) that returns a string (array of characters) each time it is called.

• Define a new data type. Make word a new string type returned by Get_Next_Word. Its structure is visible, but the compiler will detect operations on words that should only be performed on strings and vice versa.

• Encapsulate the functionality for dealing with words in a package. The package will make visible (export) a data type (word) and one or more subprograms for obtaining and processing them. This approach has four further options.

1. Publish the representation for words by using an existing Ada data type like string.
2. Use the Ada Private keyword to hide the representation of the new type but allow assignment and equality tests on words. If you use an unconstrained array type as the representation, you must publish this fact in the public part of the package specification so package users can indicate the length of the words when they are declared.
3. Use the Ada Limited Private keywords to hide the representation and prevent even the use of assignment and equality tests.
4. Use an Access type (pointer) to hide even the knowledge that an array is holding the words.

• Completely hide the data type. No type is exported. The Get_Next_Word subprogram reads and parses the next word but returns nothing. Other subprograms in the package assume the existence of a current word and perform their analysis functions on it.

The design of Ada provides considerable freedom along this range of design decisions. Most languages allow only a more restricted set of choices, either because they do not support encapsulation at all or because they include features intended to support a particular style of encapsulation to the exclusion of others.

in languages provide evidence of the need to express design ideas in code. Semantic relationships from database theory support the modeling of information structures from a variety of fields. Examination of tools used for reverse engineering and maintenance indicate decisions that have been found useful in understanding existing programs. The box on p. 47 describes several mechanisms for implementing design decisions in current languages.

Composition and decomposition. Probably the most common design decision made when developing a program is to split it into pieces. You can do this, for example, by breaking a computation into steps or by defining a data structure in terms of its fields. Introducing a construct and then later decomposing it supports abstraction by letting you defer decisions and hide details. You manage complexity by using an appropriate name to stand for a collection of lower level details.

If you take a top-down approach to design, you decompose a program into pieces. If you take a bottom-up approach, you compose a program from available subcomponents. Regardless of the approach, the result is that a relationship has been established between an abstract element and several more detailed components.

Data and control structures are language features that support these decisions. For example, a loop is a mechanism for breaking a complex operation into a series of simpler steps. Likewise, arrays and record structures are ways to collect related data elements into a single item. Of course, building an expression from variables, constants, and operators is an example of composition. So too is building a system from a library of components.

Encapsulation and interleaving. Structuring a program involves drawing boundaries around related constructs. Well-defined boundaries or interfaces limit access to implementation details while controlling clients' access to functionality. Encapsulation, abstract data types, and information hiding are all related to this concept.

Encapsulation is the decision to gather selected parts of a program into a component (variously called a package, cluster, or module.) You restrict the component's behavior by a protocol or interface so other parts of the system can interact with the component only in limited ways.

Encapsulation is a useful aid to both program comprehension and maintenance. A decision to encapsulate the implementation of a program component reflects the belief that the encapsulated construct can be thought of as a whole with a behavior that can be described by a specification that is much smaller than the amount of code in the component. If the component hides the details of a major design decision, side effects of the change are limited when that decision is altered during later maintenance. The box at left presents some alternative approaches to encapulsation in Ada.

The alternative to encapsulation is interleaving. It is sometimes useful, usually for efficiency, to intertwine two computations. For example, it is often useful to compute the maximum element of a vector as well as its position in the vector. These could be computed separately, but it is natural to save effort by doing them in a single loop. Interleaving thus makes the resulting code harder to understand and modify. Martin Feather has described how transformational programming can systematically perform interleaving.[3]

Generalization and specialization. One of the most powerful features of programming languages is their ability to describe a whole class of computations using a subprogram parameterized by arguments. Although programmers usually think of procedures and functions as abstractions of expressions, the ability to pass arguments to them is really an example of generalization. The decision of which aspects of the computation to parameterize is one of the key architectural decisions made during software design.

Generalization is a design decision in which you satisfy a program specification by relaxing some of its constraints. For example, you might require a program to compute the logarithm of a limited set of numbers. You could satisfy the requirement by providing access to a general-purpose library function for computing logarithms. The library function could compute logarithms of the complete set of required numbers as well as many others. The decision to use the library function is thus a generalization decision.

You can parameterize abstractions other than numerical computations. For example, Ada provides a generic facility that lets data types and functions parameterize packages and subprograms. Many languages provide macro capabilities that parameterize textual substitutions. For example, variant records in Pascal and Ada and type unions in C use a single, general construct to express a set of special cases, possibly depending on the value of a discriminant field.

Another example of generalization is the use of interpreters for virtual machines. It is often useful for a designer to introduce a layer of functionality controlled by a well-defined protocol. You can think of the protocol as the language for the virtual machine implemented by the layer. The decision to introduce the protocol reflects the desire to provide more generality than a set of disparate procedures would offer.

Specialization is a design decision related to generalization. It involves replacing a program specification with a more restricted one. You can often optimize an algorithm based on restrictions in the problem domain or in the language facilities. Although these optimizations can dramatically improve performance, they lengthen the program size and make it harder to understand.

Specialization also occurs in the early stages of design. Specifications are often expressed in terms of idealized objects like infinite sets and real numbers. But actual programs have space and precision limitations. Thus, a program is necessarily a special case of a more general computational entity.

In object-oriented languages like Smalltalk and C++, the designer is provided with a collection of class definitions. A class provides an implementation for objects that belong to it. A new class inherits the common functionality from its more general predecessor. Knowledgeable developers can quickly implement new classes by specializing existing classes.

Generalization and specialization decisions have long-term implications on the program being developed. It is easier to reuse or adapt a generalized component than a restricted one. But generality has a cost: Generalized components may be less efficient than specially tuned versions. Moreover, it often takes more effort to test a component intended for wide application than to test its more specific counterpart.

It is easier to reuse or adapt a generalized component than a restricted one. But generality has a cost: Generalized components may be less efficient than specially tuned versions.

Representation. The selection of a representation is a powerful and comprehensive design decision. You use representation when one abstraction or concept can better express a problem solution than another. It may be better because the target abstraction more ably captures the sense of the solution or because it can be more efficiently implemented on the target machine. For example, you may choose a linked list to implement a push-down stack. Or you may use bit vectors to represent finite sets. Representation is the decision to use one construct in place of another functionally equivalent one.

Representation must be carefully distinguished from specialization. If you implement a (possibly infinite) push-down stack with a fixed-length array, you have made two decisions. The first decision is that a bounded-length stack will work for this program. This is a specialization decision. The second is that the bounded stack can be readily implemented by a fixed-length array and an index variable. This is a representation choice.

When you keep in mind the distinction between specialization and representation, you can see representation to be a flexible and symmetrical decision. In one context, it may be appropriate to represent one construct by another. But you might use the inverse representation in a different context. For example, operations on vectors are usually implemented by a loop, but if using vector-processing hardware, the compiling system may invert the representation to reconstruct the vector operation.

Another example of representation comes from the early stages of design. Formal program specifications are often couched in terms of universal and existential quantification like "All employees who make more than $50,000 per year ... ." Programmers typically use loops and recursion to represent these specifications.

Data and procedures. Variables are not necessary to write programs, since you can always explicitly recompute values. Program variables have a cost: the amount of effort required to comprehend and modify a program. But they can improve the efficiency of the program and, by a judicious choice of names, clarify its intent.

You must be aware of the invariants relating the program variables when you insert statements into a program. For example, suppose a maintenance programmer is investigating a loop that reads records from a file and counts the number of records read. The programmer has been asked to make the loop disregard invalid records. Because the counter is used to satisfy design dependencies between this loop and other parts of the program, he must modify the semantics of the counter.

The programmer must choose from three alternatives: counting the total number of records, counting the number of valid records, or doing both. To make the correct choice, he must determine how the counter is used later in the program. In this case, he can replace references to variables with the computations that produced their most recent values. He can rearrange the resulting statements to reconstruct the high-level operations applied to the file. Having done this, the programmer can confront the semantic problems raised by the distinction between valid and invalid records. Once he has solved those semantic problems, he can again interleave the components by reintroducing assignment statements.

The introduction of variables constrains the sequence in which computations may be made. This increases the possibility of errors when modifications made during maintenance accidently violate an implicit design dependency such as causing variables to be computed in the wrong order.

The alternative to introducing a variable is to recompute values when they are needed. This sometimes makes a program more readable, since a reader does not have to search the program for the declaration and assignments to a variable but can directly use local information. Optimizing compilers often reduce the cost of recomputation, particularly where constant expressions are involved.

The decision to repeat a computation or to save the result of the computation in a variable reflects the deeper concept of the duality of data and procedure. The implementation of a finite-state machine is an example of where the data/procedure decision is apparent.

In the data-oriented approach, possibilities for the machine's next state are recorded in a two-dimensional array, often called the next-state table. Alternatively, you can directly compute the next-state information in code for each of the states. Although this may seem unusual, it is exactly the same technique used to speed up lexical analyzers: Token classes are first represented as regular expressions and then as states in a state machine. The states are then compiled directly into case/switch statements in the target language. The reason to do this is efficiency: In the procedural version, it avoids the cost of indexing into the array.

Function and relation. Logic languages let you express programs as relations between sets of data. For example, sorting is described as the relationship of two sets, both of which contain the same members, one of which is ordered. In Prolog, this might be described by the rule

```
sort(S1,S2) :-
   permutation(S1,S2),ordered(S2).
```

If *S*1 is given as input, a sorted version *S*2 is produced. But if, instead, an ordered version *S*2 is provided, unordered permutations are produced in *S*1. You can leave the decision about which variable is input and which is output to the user at runtime instead of to the developer at design time.

Formal functional specifications are often nondeterministic about the relation's direction of causality. If there is a preferred direction, the designer may use a function instead of a relation to express it. But this may reflect an implementation bias rather than a requirement.

Even in Prolog it may be impossible, for any given problem, to write a set of rules that works equally well in both directions. Thus, the designer must select the preferred direction of causality - which variables are input and which are output.

Of course, more traditional languages do not support nondeterministic relationships. Even in Prolog it may be impossible, for any given problem, to write a set of rules that works equally well in both directions. Thus, the designer is usually responsible for selecting the preferred direction of causality — which variables are input and which are output.

An alternative approach is to provide separate functions that support both directions. For example, in a student-grading system it may be useful to provide a function that, when given a numeric grade, indicates the percentage of students making that grade or higher. It may also be valuable to provide the inverse function that, when given a percentage, returns the numeric grade that would separate that proportion of the students.

Finding decisions

Maintenance and reuse require the detection of design decisions in existing code, which is a part of reverse engineering. Because reverse engineering is the process of constructing a higher level description of a program from a lower level one, this typically means constructing a representation of a program's design from its source code. The process is bottom-up and incremental; you detect low-level constructs and replace them with their high-level counterparts. If you repeat this process, the overall architecture of the program gradually emerges from the language-dependent details.

The program in Figure 1 is taken from a paper by Victor Basili and Harlan Mills[4] in which they use flow analysis and techniques from program proving to guide the comprehension process and document the results. It is a realistic example of production software in which design decisions can be recognized. (Basili and Mills obtained the program from a book by George Forsythe and colleagues.[5])

The real function Zeroin finds the root of a function, *F*, by successively reducing the interval in which it must occur. It does this by using one of several approaches (bisection, linear interpolation, and inverse quadratic interpolation), and it is the interleaving of the approaches that complicates the program.

Interleaving program fragments. A casual examination of the program indicates that it contains two Write statements that provide diagnostic information when the program is run. In fact, these statements display the progress that the program makes in narrowing the interval containing the root. The execution of the Write statements is controlled by the variable *IP*. *IP* is one of the program's input parameters, and an examination of the program indicates that it is not altered by the program and has no other use.

This leads to the conclusion that you can decompose the overall program into two pieces: the root finder and the debugging printout. To make the analysis of the rest of the program simpler, you can remove the diagnostic portion from the text being considered. This means removing

```
001        REAL FUNCTION ZEROIN (AX,BX,F,TOL,IP)
002        REAL AX,BX,F,TOL
003 C
004 C
005        REAL A, B, C, D, E, EPS, FA, FB, FC, TOL1, XM, P, Q, R, S
006 C
007 C      Computer EPS, The RelativeMachine Precision
008 C
009        EPS = 1.0
010     10 EPS = EPS/2.0
011            TOL1 = 1.0 + EPS
012        IF (TOL1 .GT. 1.0) GO TO 10
013 C
014 C      Initialization
015 C
016        IF (IP .EQ. 1) WRITE (6,11)
017     11 FORMAT('THE INTERVALS DETERMINED
               BYZEROIN ARE')
018        A = AX
019        B = BX
020        FA = F(A)
021        FB = F(B)
022 C
023 C      Begin Step
024 C
025     20 C = A
026        FC = FA
027        D = B - A
028        E = D
029     30 IF (IP .EQ. 1) WRITE (6,31) B, C
030     31 FORMAT (2E15.8)
031        IF (ABS(FC) .GE. ABS(FB)) GO TO 40
032        A = B
033        B = C
034        C = A
035        FA = FB
036        FB = FC
037        FC = FA
038 C
039 C      Convergence Test
040 C
041     40 TOL1 = 2.0*EPS*ABS(B) + 0.5*TOL
042        XM = .5*(C-B)
043        IF (ABS(XM) .LE. TOL1) GO TO 90
044        IF (FB .EQ. 0.0) GO TO 90
045 C
046 C      Is Bisection Necessary
047 C
048        IF (ABS(E) .LT. TOL1) GO TO 70
049        IF (ABS(FA) .LE. ABS(FB)) GO TO 70
050 C
051 C      Is Quadratic Interpolation Possible
052 C
053        IF (A .NE. C) GO TO 50
054 C
055 C      Linear Interpolation
056 C
057        S = FB/FA
058        P = 2.0*XM*S
059        Q = 1.0 - S
060        GO TO 60
061 C
062 C      Inverse Quadratic Interpolation
063 C
064     50 Q = FA/FC
065        R = FB/FC
066        S = FB/FA
067        P = S*(2.0*XM*Q*(Q-R) - (B-A) * (R-1.0))
068        Q = (Q-1.0)*(R-1.0)*(S-1.0)
069 C
070 C      Adjust Signs
071 C
072     60 IF (P .GT. 0.0) Q = -Q
073        P = ABS(P)
074 C
075 C      Is Interpolation Acceptable
076 C
077        IF ((2.0*P) .GE. (3.0*XM*Q - ABS(TOL1*Q)))
               GO TO 70
078        IF (P .GE. ABS(0.5*E*Q)) GO TO 70
079        E = D
080        D = P/Q
081        GO TO 80
082 C
083 C      Bisection
084 C
085     70 D = XM
086        E = D
087 C
088 C      Complete Step
089 C
090     80 A = B
091        FA = FB
092        IF (ABS(D) .GT. TOL1) B = B + D
093        IF (ABS(D) .LE. TOL1) B = B + SIGN(TOL1,XM)
094        FB = F(B)
095        IF ((FB*(FC/ABS (FC))) .GT. 0.0) GO TO 20
096        GO TO 30
097 C
098 C      Done
099 C
100     90 ZEROIN = B
101        RETURN
102        END
```

Figure 1. Realistic example of production software in which you can recognize design decisions.

lines 016, 017, 029, and 030 and modifying line 001 to remove the reference to *IP*.

The removed lines may themselves be analyzed. In fact, the job of producing the debugging printout has been decomposed into two tasks. The first produces a header line, and the second prints a description of the interval on every iteration of the loop.

Representing structured control flow. Basili and Mills begin their analysis by examining the program's control flow. In fact, the version of Fortran used in this program has a limited set of control structures that forces programmers to use Goto statements to simulate the full range of structured-programming constructs. In Zeroin, for example, lines 010-012 implement a Repeat-Until loop, lines 031-037 are an If-Then statement, and lines 050-068 are an If-Then-Else statement. These lines are the result of representation decisions by the original developer. You can detect them with straightforward analysis like that typically performed by a compiler's flow-analysis phase.

This program illustrates another technique to express control flow. In several cases (lines 043-044, 048-049, and 077-078), an elaborate branch condition is broken up into two consecutive If statements, both branching to the same place.

Each pair could easily be replaced by a single If with multiple conditions, thus further simplifying the program's control-flow structure at the expense of complicating the condition being tested.

Interleaving by code sharing. Further analysis of the control flow of the program indicates that lines 085 and 086 make up the Else part of an If-Then-Else statement. Moreover, these lines are branched into from lines 048 and 049 — the two assignment statements are really being shared by two parts of the program. Two execution streams are interleaved because they share common code. Although this makes the program shorter and assures that both parts are updated if either one is, it makes understanding the program structure more difficult.

To express the control flow more cleanly, you must construct a structured version. This requires that the shared code be duplicated so each sharing segment has its own version. If the common statements were more elaborate, you could introduce a subroutine and call it from both sites. As it is, it is simple here to duplicate the two lines producing two properly formed conditional constructs.

Data interleaving by reusing variable names. An unfortunately common programming practice is to use the same variable name for two unrelated purposes. This naturally leads to confusion when trying to understand the program. You can think of it as a kind of interleaving where, instead of two separable segments of code being intertwined at one location in the program, two aspects of the program state share the use of the same identifier.

This occurs twice in Zeroin, with the identifiers Tol1 (in lines 011-012 and in the rest of the program) and *Q* (on line 064 through the right side of line 068 and in the rest of the program, including the left side of line 068). Compilers can detect instances of this practice with dataflow analysis.

Generalizing interpolation schemes. Zeroin has two sections of code that use alternative approaches to compute the values of the same set of variables. Both lines 057-059 and 064-068 compute the values of the variables *P* and *Q*. The determination of which approach to use is based on a test made on line 053. This is an example of specialization.

You can replace both computations and the test conceptually by a more general expression that computes *P* and *Q* based on the current values of the variables *A*, *B*, *C*, *FA*, *FB*, *FC*, and *XM*. This has the further benefit of localizing the uses of the variables *S* and *R* in the new expression.

There are really several design issues involved here. First, both code segments result from the decomposition of the problem into pieces expressed by a series of assignment statements. Then, the realization that both segments are specializations of a more general one lets the details of the individual cases be hidden. This, in turn, makes the code shorter and easier to understand.

Variable introduction. A common programming practice is to save the result of a computation to avoid having to recompute the same value later. If the computation is involved, this practice can result in

Design-decision projects

The design-decision concept plays a central role in several research efforts at Georgia Institute of Technology. These projects span a considerable range of applications, including specification-driven software reuse, a generalization of the application-generator concept, reverse engineering of Cobol programs, and transformation-based compiler construction.

These projects all depend (to varying degrees) on the existence of a program representation that expresses the link between specification, design, and code. This representation must also provide some way to record the rationale behind the transformations that create the design and code from the specification.

The specification-based reuse project is driven by the idea that reusing specification-level components makes it easier to customize components to facilitate their reuse. In this project, a component is considered to consist of a specification, one or more versions of code that implements the specification, and scripts that describe the transformations necessary to derive the code version(s) from the specification. You can immediately reuse a component whose specification matches a need if the set of implementation decisions captured by a script is suitable for the reuse context. If not, you might choose alternative decisions, thus creating a slightly modified script. One of this project's long-term goals is to build an environment to automatically replay a script of transformations.

The application-generator project uses this same technology. You can think of an application generator as an environment for producing variants of a program. The generator predefines the program's basic structure. The specification of the variants is, in effect, a specification of a set of decisions that determine certain implementation details. Each component of such a specification either chooses among alternatives or fills in blanks in the standard script that represents the application-generation process. The advantage of this approach is that it supports the construction of a wide variety of application generators by providing an underlying conceptual basis for their design.

The Cobol reverse-engineering project is an experiment in translating Cobol programs to Ada. The planned approach is to derive a functional description and a high-level design from examining the code and available documentation. The reverse-engineering process relies heavily on the design-decision recognition process described in the accompanying article, thus providing empirical validation of our model if it is successful. The project will also explore the representation issues in linking the new descriptions to the original code and representing the connections between them with design decisions.

The transformation-based compiler-construction project is exploring the derivation of software in this well-understood domain. You can think of the alternative design decisions, like the compiler's structure or the target machine's instruction set, as alternative transformation paths in the derivation of a compiler from its specification. This project's short-term goal is to guarantee the correctness of code generation, given a formal semantic specification for the source language and a suitable target-machine description.

a significant savings at runtime with a modest cost.

Zeroin contains several instances of this practice. They made a concerted effort to save the results of calls to the user-supplied function, *F*, in the variables *FA*, *FB*, and *FC*. Because *F* may be arbitrarily complex, this practice may be the most important determinant of Zeroin's ultimate efficiency.

An examination of the program reveals that *FA*, *FB*, and *FC* always contain the results of applying *F* at the points *A*, *B*, and *C*, respectively. From the standpoint of understanding the algorithm, these three additional variables do not provide a significant abstraction — to the contrary, they require a nontrivial effort to understand and manipulate. Replacing them with their definitions makes the resulting program easier to understand.

When readability is the goal, there are two factors to be weighed in deciding whether to write the program with a variable name or to replace it with its value. One factor is that each new variable places a burden on the person trying to understand the program, since he must read the variable and understand and confirm its purpose. The other factor is that variables can be valuable abbreviations for the computation that they replace. It is easier to understand a variable with a carefully chosen name than the complex expression it represents.

In the case of Zeroin, the variables *FA*, *FB*, and *FC* provide little in the way of abstraction. *P* and *Q*, on the other hand, abbreviate significant computations, although without the benefits of mnemonic names. *XM* lies somewhere in the middle.

Generalizing interval computation. Now that the recognition of some intermediate decisions has clarified the program structure, you can make the same sort of observation about lines 048-086. They assign values to the variables *D* and *E* based on the values of the variables *A*, *B*, *C*, *D*, *E*, *F*, Toll, and *XM*. The fact that the list of variables is so long indicates that this segment is highly interleaved with the rest of the program. Nevertheless, it is helpful to indicate that the only explicit effect of these lines is to set the two variables.

There are also several instances of specialization in these lines. Lines 079-080 and 085-086 are selected based on the tests on lines 077-078. Likewise, lines 082-086 and 050-081 are special cases selected on the basis of the tests on lines 048-049.

```
001      Real Function Zeroin (AX,BX,F,TOL)
024 C
028      initialization
    C
         loop
            conditional adjustment 1
038 C
043         if (close enough to final answer)
                     return(B)
045 C
092         compute new value of B
    C
            conditional adjustment 2
096      endloop
```

Figure 2. Program architecture after analyzing its structure.

Program architecture. Once you have performed this analysis, you can appreciate the program's overall structure. Based on the test in line 044, you can see that the program uses the variable *B* to hold approximations of the root of the function. *B* is modified on lines 092-093 by either *XM* or *D*. The sections on lines 025-028 and 031-037 act as adjustments in special situations.

Another conclusion that is now apparent is that *A* gets its value only from *B*, while *C* gets its value only from *A*. Thus, *A*, *C*, and *B* serve as successively better approximations to the root. In fact, except under special circumstances, *A* and *C* have identical values. Likewise, *E* normally has the same value as *D*. Figure 2 shows the resulting program architecture.

Representing decisions

It is not sufficient to simply recognize design decisions in code. Once recognized, you must organize the decisions so maintenance programmers and reuse engineers can use them. The organization chosen serves as a representation for design information.

There are many methods for designing software and many representations for the intermediate results. Typically, you use several during program design, some during the architectural stages and others during low-level design. You may use still others during maintenance if there is a separate maintenance staff. It may thus be difficult to recreate and reuse the original representation.

A usable representation for design information must be easy to construct during development and easy to reconstruct during reverse engineering. Once constructed, it must facilitate queries and report generation to support maintenance. It must provide a way to attach available documentation. Also, it must support automation.

The representation must be formal enough that its components can be automatically manipulated. For example, it is desirable to be able to determine if a previously developed partial description of a software component can be reused in a new situation.

A representation for design information must let all types of design information be attached. This includes high-level specifications, architectural overviews, detailed interfaces, and the resulting source code. It is also desirable that the representation support requirements tracing, informal annotations, and version information.

Several approaches to organizing design information have been proposed.

Ted Biggerstaff's Desire system[6] is concerned with relating code fragments to information from the problem domain. Reuse will be facilitated if a new problem's requirements can be easily matched against a description of existing software.

Mark Blackburn's work[7] is also concerned with reuse. He proposes a network of design information where fragments are connected by one of two relationships, either "is-decomposed-into" (decomposition) or "is-a" (specialization).

Derek Coleman and Robin Gallimore have reported on FPD,[8] a framework for program development. Arcs in their network model correspond to refinements steps taken during the design. Each refinement engenders a proof obligation to guarantee the correctness of the step taken.

Maintenance and reuse require of their practitioners a deep understanding of the software being manipulated. That understanding is facilitated by the presence of design documentation. Effective documentation should include a description of the software's structure and details about the decisions that led to that structure.

Design decisions occur where the abstract models and theories of an application domain confront the realities of limited machines and imperfect languages. If the design decisions can be reconstructed, there is greater hope of being able to maintain and reuse the mountains of undocumented software confronting us all. ❖

References

1. D.E. Webster, "Mapping the Design Representation Terrain: A Survey," Tech. Report STP-093-87, Microelectronics and Computer Technology Corp., Austin, Texas, July 1987.
2. E. Soloway et al., "Designing Documentation to Compensate for Delocalized Plans," *Comm. ACM*, Nov. 1988, pp. 1,259-1,267.
3. M.S. Feather, "A Survey and Classification of Some Program-Transformation Approaches and Techniques," in *Program Specification and Transformation*, L.G.L.T. Meertens, ed., North Holland, Amsterdam, 1987, pp. 165-195.
4. V.R. Basili and H.D. Mills, "Understanding and Documenting Programs," *IEEE Trans. Software Eng.*, May 1982, pp. 270-283.
5. G. Forsythe, M. Malcolm, and M. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, Englewood Cliffs, N.J., 1977.
6. T.J. Biggerstaff, "Design Recovery for Maintenance and Reuse," *Computer*, July 1989, pp. 36-49.
7. M.R. Blackburn, "Toward a Theory of Reuse Based on Formal Methods," Tech. Report SPC-TR-88-010, Version 1.0, Software Productivity Consortium, Herndon, Va., April 1988.
8. D. Coleman and R.M. Gallimore, "A Framework for Program Development," *Hewlett-Packard J.*, Oct. 1987, pp. 37-40.

Spencer Rugaber is a research scientist with the Software Engineering Research Center and the School of Information and Computer Science at Georgia Institute of Technology. His research interests include software maintenance and reverse engineering.

Rugaber received a BS in engineering and applied science and a PhD in computer science from Yale University and an MS from Harvard in applied math. He is a member of the ACM.

Stephen B. Ornburn is a PhD candidate and research assistant in Georgia Institute of Technology's School of Information and Computer Science. His dissertation research investigates the application of transformational programming techniques to problems in the reuse of software. Other research interests include the design of fault-tolerant software and the economic value of information systems. Before pursuing his PhD, he was an industrial engineer for General Foods.

Ornburn received a BS in industrial engineering and management science from Northwestern University and an MS in computer science from Georgia Institute of Technology. He is a member of the IEEE Computer Society and ACM.

Richard J. LeBlanc, Jr., is a professor in Georgia Institute of Technology's School of Information and Computer Science. His research interests include language design and implementation, programming environments, and software engineering. His research work involves applying these interests to distributed systems. He is also interested in specification-based development methodologies and tools.

LeBlanc received a BS in physics from Louisiana State University and an MS and a PhD in computer sciences from the University of Wisconsin at Madison. He is a member of the ACM, IEEE Computer Society, and Sigma Xi.

Address questions to the authors at Software Engineering Research Center, Georgia Institute of Technology, Atlanta, GA 30332-0280.

FREE CATALOG

1-800-547-5444

In Canada, Call Toll Free 1-800-387-2173

Call today, and get the Inmac Computer Products Catalog!

- Guaranteed Delivery
- Over 3000 Products To Choose From
- Instant Credit
- 45 Day Product Trial

☑ *YES, Please send my first Inmac catalog today!*

Name	Title	
Company		
Street Address/P.O. Box		
City	State	Zip
( ) Area Code Phone No.		5566

inmac® *"Your Worldwide Source for Computer Supplies, Furniture, and Data Communications Products."*

Mail To: Inmac 2465 Augustine Drive, Santa Clara, CA 95054

Reader Service Number 5