RECOGNITION AND REPRESENTATION OF DESIGN DECISIONS

Stephen B. Ornburn*
Spencer Rugaber
School of Information and Computer Science and
Software Engineering Research Center
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

Abstract

Reverse engineering is the process of constructing a higher level description of a program from a lower level one. Typically, this means constructing a representation of the design of a program from its source code. Design is the process of making decisions from among a set of possible alternatives. Usually the selection is made based on non-functional considerations such as efficiency or reliability. Thus, reverse engineering requires the recognition of design decisions in the source code. But what is a design decision? This paper describes a characterization of design decisions based on the analysis of source code. Once design decisions are recognized, they can be organized into a structure that can be used to understand the program and to perform maintenance tasks such as enhancement or bug fixing.

*support received from NSF grants ....

1 REVERSE ENGINEERING AND DESIGN DECISIONS

1.1 Need for Reverse Engineering

Reverse engineering is the process of constructing a higher level description of a program from a lower level one. Typically, this means constructing a representation of the design of a program from its source code. A high-level representation hides the details of the implementation while emphasizing the structural and semantic relationships among the components. Reverse engineering is not an end in itself. It supports software maintenance activities such as program comprehension, correction, documentation, and enhancement. The information captured during the reverse engineering process should be organized in a manner to support these activities.

In the ideal world, there would be no need for reverse engineering. All of the steps taken and decisions made during initial development would be captured along with their rationales. Included with this information would be the reasons for not making alternative choices. When the program is later changed during the maintenance phase, the original development process could be partially replayed, including the updating of intermediate documentation, before the code is ultimately modified.

In real life situations, however, intermediate decisions, representations, and rationale may not have been captured. Even if captured, they may not have been kept up-to-date when subsequent changes were made. Modifications made without benefit of this information often suffer from a limited view of the overall goals and architecture of the program. This leads to a deterioration in the structure and coherence of the program. By creating or recreating the intermediate and high-level representations, reverse engineering tries to overcome these problems.

1.2 Design Decisions

Software design is the process of taking a functional specification and a set of non-functional constraints and producing a description of an implementation from which source code can be developed. Design can be divided into high-level design, often called architectural design, and low-level design, where the concerns are with data structures and algorithms.

Design involves making explicit choices among alternatives. These design decisions are often involve engineering trade-offs. Too often, however, the alternatives that are considered and the rationale for the final choice are lost. Methodologies and systems are now being developed that support and document the decision making process. Once such systems are in common use, the reverse engineering problem will be much reduced.

One reason design information is lost is that the design representations currently in use are not expressive enough. While they are adequate for describing the cumulative results of a set of decisions, particularly in regard to the structure of components and how they interact, they do not have the resolving power necessary to represent the incremental changes that come with individual design decisions. Also, they fail to describe the process by which decisions were reached, including the relevant requirements and the relative merits of the alternative choices.

What is a design decision? It is a choice made by the programmer about the internal structure of the program. Within the framework of this paper, a design decision is a transformation which maps a high-level version of a program onto a lower level one. A high-level version specifies the dependencies among components. A low-level version, on the other hand, shows how the implementation will implicitly satisfy these dependencies.

The differences between a high and low-level version of a program are the result of transformations. These transformations can change the the decomposition and arrangement of components, their interconnections, or the algorithms and data structures they use. In all cases, however, the version that results must satisfy any design dependencies explicitly stated in the high-level design.

Design dependencies may be of many varieties, and several examples are provided here. Design dependencies may relate the bounds of a for-loop to the size of an array. They may require that the conditions in a conditional statement cover all of the possible situations that may arise. They may require that the domain of one function include all of the values produced by another. They may even require that the same set of procedure names be used for several data types in order to show that the types taken together implement a particular data abstraction.

The detection of design dependencies among components is the key to reverse engineering. A dependency reflects the fact that a high-level design component fhas been represented by a combination of lower level design elements. The lower level elements need not be contiguous. They may distributed throughout the text. A component whose elements have been distributed is referred to as a *delocalized* component[5]. If the dependencies among the elements of a delocalized component have not been appreciated by maintenance programmers, bugs may be introduced when a modification is made. The changes which must be made to ensure the dependencies are satisfied can cause the the structure of the program to deteriorate.

1.3 Requirements for a Design Representation

There are numerous methods for designing software and numerous representations for the intermediate results[4]. Typically, several are used during the design of a program, some during the architectural stages and others during low-level design. Still others may be used during the maintenance stage if the original developers have given way to a separate maintenance staff. It may consequently be difficult to recreate and reuse the original representation.

Thus, reverse engineers are faced with constructing a representation for the design information of a program that may not be the same as that used to develop it in the first place. The representation must be adequate not only to express the original, presumably well structured, software but also later modifications made to the program.

A usable representation for design information must be easy to construct from the source code. Once constructed, it must facilitate queries and report generation in order to support software maintenance activities. It must provide a mechanism for attaching available documentation. Also, it must support automation. In particular, 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 is reusable in a new situation. A representation for design information must allow all types of design information to 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 versioning information.

The problem that this paper addresses is the design of a methodology-independent organization for design information that can be readily constructed from source code. Its approach is to concentrate on the question of what is a design decision. As the reverse engineer recognizes design decisions in the source code, they are collected into a repository that can later be perused or transformed during software maintenance.

1.4 Organizing Design Decisions

Design decisions can be broken down into categories roughly corresponding to the abstraction mechanisms and modeling constructs used to realize them. For example, *specialization* is a type of refinement often seen in object-oriented programming language where an already existing component is refined to handle a special case situation. *Representation* is used when a decision is made to select an appropriate programming language construct with which to implement a specific data structure or algorithm. A catalog of such design decisions allows the reverse engineer, supported by automated analysis tools, to recognize and organize design information.

This design information can be used for a variety of purposes. It can be used in order to gain a greater understanding of the structure of the software. For example, if only *decomposition* decisions are viewed, then the result is a functional decomposition of the design. If annotations are attached, the maintainer can learn the reasoning behind a decision. Because all stages of the design process are available, a programmer interested in reusing the design can locate and extract the part that matches the programmer's needs.

2 CHARACTERIZATION OF DESIGN DECISIONS

2.1 Overview

During the course of development of a program, many decisions are made. Some of these are explicit, but many reflect implicit assumptions based on experience or judgement concerning the "best" course to follow. In order to effectively maintain an existing system, it is essential for the maintenance programmer to sustain previously made decisions unless the reasons for the decision have also changed. In order for this to be done effectively, the decisions must be recognized and understood.

Design decisions range from low-level concerns necessitated by the programming language used to structural considerations related to the components of the system and their interactions. Traditional design methods recognize the differences between these two, but the borderline is not well defined.

Design decisions can be classified according to the language constructs used to realize them in an implementation language. For example, a programmer may decide that a computation will be performed often enough that its results should be saved and reused. This typically involves the introduction of a variable and its assignment as the result of the evaluation of an expression. The name of the variable can then be referred to later in the program.

2.2 Categories of Design Decisions

2.2.1 The Data/Procedure Decision

The issue of whether to repeat a computation or to introduce state into a program by introducing a variable reflects the deeper concept of the duality of data and procedure. Variables are not necessary in order to write programs; values can always be recomputed. Program variables have a cost in terms of the amount of effort required to comprehend and modify a program. On the other hand, they can serve to improve the efficiency of the program and, by a judicious choice of names, serve to clarify its intent.

The introduction of state constrains the sequence in which computations may be made. This increases the possibility of errors when variables are computed in the wrong order or when modifications made during maintenance accidently violate an implicit ordering constraint.

The dual of state introduction is recomputation. This is sometimes used to make a program more readable. The reader does not have to search the program for the declaration and assignments to a variable but can use local information directly. Modern optimizing compilers can reduce the cost associated with recomputation, particularly where constant expressions are involved.

The programmer must be aware of how the data flows though the program. Modifications made during maintenance that change the sequence in which two statements are executed can introduce a bug if one statement changes the value of a variable referenced by the other. By undoing the assignment statements and introducing redundant computations the programmer can avoid sequencing problems when reordering statements. Once the new sequence has been established, assignment statements can be reintroduced.

Programmers must also be aware of the invariants relating the program variables when inserting statements into the program. For example, suppose a programmer is maintaining a loop that reads records from a file and keeps count of 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 components, the programmer must modify the semantics of the counter. The programmer must choose from among three alternatives: counting the total number of records, counting the number of valid records, or doing both. To make the correct choice, the programmer must determine how the counter is used later in the program. In this case, the programmer can undo assignment statements and rearrange statements in order 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 those semantic problems have been solved, components can again be delocalized and assignment statements reintroduced.

2.2.2 Generalization/Specialization

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

Other abstractions may also be parameterized. Ada provides a *generic* facility that allows types and functions to parameterize packages and subprograms. Many languages provide macro capabilities that parameterize textual substitutions.

The dual of generalization is specialization. This is often seen in object-oriented programming languages where an inheritance facility can be used to quickly build specialized classes from more generalized predecessors. Of course, the decision to use an existing procedure in a traditional language is also an example of specialization when arguments are passed to it. Nor is this issue restricted to the procedural domain. Variant records in languages such as Pascal are an example of the use of a single general construct to express a set of special cases depending on the value of a discriminant field.

Another example of generalization concerns interpreters and virtual machines. It is often useful for a designer to introduce a layer of functionality that is controlled by a well-defined protocol. The protocol can be thought of as the programming 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 provide.

A more subtle example of these issues concerns specialization. Often algorithms can be optimized based on restrictions in the problem domain or facilities of the programming language. Although these optimizations can dramatically improve the performance of the algorithm, they have a cost in lengthening the program and making it harder to understand. Another manifestation of this can be seen in the early stages of the design process. Often specifications are expressed in terms of idealized objects such as infinite sets and real numbers. Actual programs have space and precision limitations. Thus a program is necessarily a special case of a more general computational entity.

When reverse engineering, useful insights can result from generalizing a component. Generalization allows the programmer to determine how the value of a parameter introduced by the generalization depends on other values used in the program. Generalization also allows the maintenance task to be specified in terms of a change in the value of a parameter. Once the maintenance task has been specified in terms of a parameter change and the dependencies among the parameters of several generalized components have been identified, then the programmer can identify the implementation changes required to complete the maintenance task.

2.2.3 Composition/Decomposition

Probably the most common design decision made when developing a program is to break it into pieces. This can be done, for example, by a breaking a computation into steps or by defining a data structure in terms of its fields. The prospect of introducing a component and then later decomposing it supports abstraction by allowing decisions to be deferred and details hidden. Complexity is managed by using a name to stand for a collection of lower level details.

The dual of decomposition is composition. One manifestation of the duality is the tension between top-down and bottom-up design methods. Composition involves composing low-level constructs into high-level abstractions. Of course, building an expression in a programming language from variables, constants, and operators is an example of composition. So too is building an algorithm from calls to a library of components.

Data and control structures are programming language features that support these decisions. For example, a loop is a mechanism for breaking a complex operation into a series of similar and simpler steps. Likewise, an array is a way of collecting similar data items into a single item.

It is not necessary for the components of a high-level construct to appear contiguously in the final program. A failure to recognize that a group of unconnected pieces of program text work together for a common purpose and represent a *delocalized* component is often a stumbling block to the program comprehension process. How a single design-level component can be distributed throughout a program text as the result of a design decision is discussed in the Section 2.2.4.

2.2.4 Encapsulation and Interleaving

Structuring a program involves drawing boundaries around related constructs. Well defined boundaries or interfaces serve to limit access to implementation details while providing controlled access to functionality by clients. The terms *encapsulation, abstract data types,* and *information hiding,* are all related to this concept.

Encapsulation is a useful aid to both program comprehension and maintenance. A decision to encapsulate the implementation of a program segment 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 total amount of code contained within the capsule. If a capsule hides the effects of a major design decision, then when that decision is altered during later maintenance, the side-effects of the change are limited.

The dual of encapsulation is interleaving. It is sometimes useful, usually for reasons of 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 once. Interleaving in this way makes the resulting code harder to understand and modify. A number of interleaving transformations have been identified [2].

A debate is currently underway concerning single and multiple inheritance in object-oriented programming language. The use of inheritance can simplify both generalization and specialization. A variant of an object is more easily constructed when most of its functionality is inherited from an already existing object. But, in nature, objects often belong to multiple classes. Some languages support multiple inheritance, where objects can inherit properties from more than one parent object. The properties of the parents are interleaved in the child. The programmer building the child is then responsible for carefully distinguishing the functionality to be used in any computation. This problem is compounded when name conflicts arise among the properties in the parent classes.

Because of the sequencing dependencies introduced, the presence of the assignment statements can make it difficult to subsequently separate the interleaved elements. Indeed, it may be necessary to undo the assignment statements before the functional components can be reassembled and encapsulated as discrete, identifiable entities. It is to be expected that the considerable effort required to reencapsulate components will be well spent. Many dependencies can be more clearly expressed at a higher level of granularity.

2.2.5 Representation

Representation is a subtle but important design decision. Representation is used when one abstraction or concept is better able to express a problem solution than another. For example, programmers often use a linked list to implement a pushdown stack. Bit vectors can be used to represent finite sets.

Representation must be carefully distinguished from specialization. If a (possibly infinite) pushdown stack is implemented by a fixed length array, then two decisions have been made. The first decision is that for the purposes of this program, a bounded length stack will serve. This is a specialization decision. Then the bounded stack can be readily implemented by a fixed length array. This is a representation choice.

A table-driven state machine is an example where a set of computations is represented by a two dimensional array. Alternatively, the array can also be represented by a procedure which computed the "stored" values. Although this may seem unusual, it is exactly the technique that is used to speed up lexical analyzer generators. Token classes are first represented as regular expressions and then state machines. The state machines are then compiled directly into case/switch statements in the target programming language. The reason for doing this is efficiency: in the compiled version the costs of indexing into the array are avoided.

When the distinction between specialization and representation is kept is mind, representation can be seen to be its own dual. For example, a designer will often implement operations on vectors by a loop. In the presence of vector processing hardware, however, 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; e.g. "All employees who make over $50,000 per year". Programming languages typically use loops and recursion to represent these specifications. Use of a stream data structure to represent a sequence of computed values is another example from this domain.

When reverse engineering, the effort is directed towards identifying the data and procedures in the program with higher level functions and abstract data types. Representation decisions have been understudied in the literature. A discussion of representation decisions as presently understood can be found in [3]. The tools developed there are appropriate for studying representation decisions when abstract values are in one-to-one correspondence with their representations. In practice, however, the programmer must be aware of a number of other possibilities.

One possibility, already discussed, involves choices such as whether a function should be represented as a data structure or a data structure represented as a function. There are also a number of alternatives for representing data structures with other data structures. For example, several high-level types may be mapped to a single low-level type, a situation known as *overloading*. On the other hand, a single high-level type is said to have a *polymorphic* representation if, as in highly optimized algorithms, it can be mapped to several distinct low-level types. This does not exhaust the range of possibilities. Structure sharing may result when a single low-level object is used to represent a set of distinct high-level objects. In such situations, maintenance tasks that at the high level may appear simple can involve considerable manipulation at the low level. Some of the hardest examples of this type of maintenance arise in the context of object-based, distributed systems: in such systems meeting performance requirements entails a number of representation decisions which violate the high-level abstractions. Specifically, in such systems violations of high-level abstractions can result from violations of atomicity, optimistic concurrency control, recovery procedures sensitive to the semantics of the objects and applications, and the replication of data and computation. Thus, object-based, distributed systems can be expected to present programmers with complex maintenance problems.

2.2.6 Function/Relation

Logic programming languages allow programs to be expressed 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 following rule.

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

If S1 is given as input, then a sorted version S2 is produced. But if an ordered version S2 is provided, then unordered permutations are produced in S1. The decision as to which variable is input and which is output can often be left up to the user at run-time instead of the programmer at design time.

Formal functional specification are often non-deterministic in this regard. If there is a preferred direction, then specification may use a function instead of a relations to express it. But this may reflect an implementation bias rather than a requirement.

Of course, empirical programming languages do not support non-deterministic relationships. Even is Prolog (the predominant logic programming language), it may be impossible 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; that is, which variables are input and which are output.

Ada makes the decision explicit with its **in**, **out**, and **inout** keywords. The dual decision 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 of value to provide the inverse function that, when given a percentage, returns the numeric grade that would separate that proportion of the students.

2.3 Programming Languages and Design Decisions

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

The languages Algol and Pascal introduced and systematized control and data structures to programming languages. 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 programming languages in the form of subroutines 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 programming languages. Programming in a functional language is difficult for a traditional programmer used to using variables.

Similarly, logic programming languages, such as 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 issues of *encapsulation* and *interleaving* are explicitly stressed in Ada. The programmer expresses the functional interface to a package in a separate construct from the implementation. Languages with multiple inheritance, such as Common Loops, must directly address how an object derived from several classes accesses properties in those parents with the same names.

Generalization/Specialization is a primary consideration in Smalltalk-80. The class hierarchy expresses how subclasses specialize their parents. Dynamic binding is used to invisibly delegate responsibility for computation to the least general class able to handle it. Of course, Ada generics support compile-time generalization of procedures to types and functional arguments.

Representation is found in most programming languages, but Clu emphasizes the distinction between *representation* and *specialization* by providing separate language constructs for expressing their behavior. Gypsy has features for describing both ideal behavior and implementation details. It supports the proof of their equivalence via semi-automated means.

2.4 Representing Design Decisions

A decision implies a choice. Hence, decisions can be thought of as the resolution of an issue by the selection of an alternative. There are two important ways of the representing design information relevant to a design decisions: as dependencies among components and as transformations. Dependencies constrain the structure of the implementation. Transformations carry out a decision to change the structure of the design. A wide variety of transformations have been described in the literature [2].

While existing characterizations of transformations are not fully satisfactory, it can be useful to classify them informally. The set of categories used here are useful when a programmer is reverse engineering a program text. Transformations can be used to change the decomposition underlying a design (i.e., breaking down or consolidating the structure of a component), to restructure a design by intertwining components (e.g., modifying the flow of data and control so as to improve an algorithm or the coupling among components), to modify the representation of data and control information, and to change the way data is stored and accessed. Changes in the way data is stored and accessed include alterations to the couplings among components and changes in how a program interacts with its state. Many of the transformations just mentioned have the effect of mapping a localized component in a high-level design to a delocalized component in a lower level design.

In addition to delocalizing components, transformations may also have other effects. It is often convenient to consider a design that presents some key components in a localized form, even if they are delocalized in the implementation. It can also be convenient to parameterize those components. Components can be parameterized for types, values, functions, and control information. Design dependencies can then be stated in terms of relationships among those parameters. Transformations, when applied, not only delocalize components, but can remove or modify design dependencies or introduce components satisfying them implicitly.

The categories used here have been synthesized from work related to the derivation of algorithms, optimization of programs, and translations among programming language paradigms such as those associated with the predicate calculus, Horn clauses, and functional, object-oriented, and imperative languages. Transformations identified in different contexts often bear a striking resemblance. Thus, there is interest in describing a more fundamental model in terms of which known transformations can be defined. There is also a conviction that the set of known transformations is not yet complete: there are categories of transformational problems that have not yet been fully explored, including some as yet only poorly conceptualized. None the less, there is a basis for optimism and value in exploring how an understanding of transformational programming could be used clarify issues in software maintenance.

3 EXAMPLES OF DESIGN DECISIONS

This section describe a realistic examples of design decisions recognized in production software. The program is taken from a paper by Basili and Mills [1] in which they use flow analysis and techniques from program proving to guide the comprehension process and document the results. The program is given in Figure 1.

```
      REAL FUNCTION ZEROIN (AX,BX,F,TOL,IP)
      REAL AX,BX,F,TOL
C
C
      REAL A, B, C, D, E, EPS, FA, FB, FC, TOL1, XM, P, Q, R, S
C
C  COMPUTER EPS, THE RELATIVE MACHINE PRECISION
C
      EPS = 1.0
   10 EPS = EPS/2.0
      TOL1 = 1.0 + EPS
      IF (TOL1 .GT. 1.0) GO TO 10
C
C  INITIALIZATION
C
      IF (IP .EQ. 1) WRITE (6,11)
   11 FORMAT('THE INTERVALS DETERMINED BY ZEROIN ARE')
      A = AX
      B = BX
      FA = F(A)
      FB = F(B)
C
C  BEGIN STEP
C
   20 C = A
      FC = FA
      D = B - A
      E = D
   30 IF (IP .EQ. 1) WRITE (6,31) B, C
   31 FORMAT (2E15.8)
      IF (ABS(FC) .GE. ABS(FB)) GO TO 40
      A = B
      B = C
      C = A
      FA = FB
      FB = FC
      FC = FA
C
C  CONVERGENCE TEST
C
   40 TOL1 = 2.0*EPS*ABS(B) + 0.5*TOL
      XM = .5*(C-B)
      IF (ABS(XM) .LE. TOL1) GO TO 90
      IF (FB .EQ. 0.0) GO TO 90
C
C  IS BISECTION NECESSARY
C
      IF (ABS(E) .LT. TOL1) GO TO 70
      IF (ABS(FA) .LE. ABS(FB)) GO TO 70
C
C  IS QUADRATIC INTERPOLATION POSSIBLE
C
      IF (A .NE. C) GO TO 50
C
C  LINEAR INTERPOLATION
C
      S = FB/FA
      P = 2.0*XM*S
      Q = 1.0 - S
      GO TO 60
C
C  INVERSE QUADRATIC INTERPOLATION
C
   50 Q = FA/FC
      R = FB/FC
      S = FB/FA
      P = S*(2.0*XM*Q*(Q-R) - (B-A) * (R-1.0))
      Q = (Q-1.0)*(R-1.0)*(S-1.0)
C
C  ADJUST SIGNS
C
   60 IF (P .GT. 0.0) Q = -Q
      P = ABS(P)
C
C  IS INTERPOLATION ACCEPTABLE
C
      IF ((2.0*P) .GE. (3.0*XM*Q - ABS(TOL1*Q))) GO TO 70
      IF (P .GE. ABS(0.5*E*Q)) GO TO 70
      E = D
      D = P/Q
      GO TO 80
C
C  BISECTION
C
   70 D = XM
      E = D
C
C  COMPLETE STEP
C
   80 A = B
      FA = FB
      IF (ABS(D) .GT. TOL1) B = B + D
      IF (ABS(D) .LE. TOL1) B = B + SIGN(TOL1,XM)
      FB = F(B)
      IF ((FB*(FC/ABS (FC))) .GT. 0.0) GO TO 20
      GO TO 30
C
C  DONE
C
   90 ZEROIN = B
      RETURN
      END
```

Figure 1

The program finds the root of a function, F, by successively shrinking 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.

Programs express the solution of a problem from an application domain within the notational constraints of a specific programming language. Hence, part of the source code expresses problem-specific details, and part is a manifestation of the idiosyncrasies and limitations of the programming language. Software maintainers can be expected to have a complete understanding of the latter, but their knowledge of the former is often limited. The examples given in this section attempts to limit the amount of domain knowledge required of the maintainer. This was also a goal of the Basili/Mills approach.

3.1 Interleaving of Program Fragments

A casual examination of the program indicates that it contains several WRITE statements that may prove useful in understanding the program. In fact, these statements display the progress that the program makes in narrowing the interval containing root. They are 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 that IP is used for no other purpose.

This leads to the conclusion that the debugging printout is a separable piece of functionality that has been interleaved into the program. To make the analysis of the rest of the program simpler, it can be removed from the text. This involves removing statements numbered 016, 017, 029, and 030 and modifying line 001 to remove the reference to IP.

The lines that have been removed are themselves analyzable. 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 out a description of the interval upon every iteration of the loop.

3.2 Representing Structured Control Flow in Fortran

Basili and Mills begin their analysis by examining the control flow of the program. This has the advantage of facilitated the confirmation of later hypotheses. 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 serve as an **if-then** statement, and lines 050-068 are an **if-then-else**. These lines are the result of representation decisions by the original programmer. They can be detected by straightforward analysis such as that typically performed by the flow analysis phase of a compiler.

Another technique for expressing control flow is illustrated in this program. In several cases (lines 43-44, 48-49, and 77-78), 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 control flow structure of the program at the expense of the condition being tested.

3.3 Interleaving by Code Sharing

Further analysis of the control flow of the program indicates that lines 085 and 086 comprise 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. That is, two execution streams are interleaved because they share common code. Although this makes the program somewhat shorter and assures that both parts are updated if either is, it makes understanding the program structure more difficult.

In order to express the control flow more cleanly, it is necessary to construct a *structured* version. This requires that the shared code be duplicated so that each of the sharing segments has its own version. If the common statements were more elaborate, a subroutine could be introduced and called from both sites. As it is, it is a simple matter to duplicate the two lines producing a properly nested conditional construct.

3.4 A Loop with Two Entrances

The bulk of the code of this program (lines 022-096) consists of a loop. The loop iterates until a root is found or the size of the interval is reduced to a user-supplied limit (lines 043-044). What is of interest from an understanding point of view is that the loop has two entrances, one at line 025 and one at line 031. The one at 025 is used in two circumstances, when the loop is first entered and when an iteration computes two new endpoints on the same side of the root. This multiple use of lines 025-028 is the result of interleaving these two computational streams.

Multiple entrances make loops hard to understand and complicate the process of confirming hypotheses based on analysis of paths through the program. There are various ways to represent the same program flow while simplifying the looping structure. For example, lines 025-028 could be duplicated and inserted after line 095 (at the same time representing 095 as an **if-then** statement). Alternatively, the test on line 095 could be moved to the top of the loop. In either case, the result is a single-entrance loop.

3.5 Data Interleaving by Reusing Variable Names

An unfortunately common practice in programs is to use the same variable name for two unrelated purposes. This naturally leads to confusion when trying to understand the program. It can be thought of 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 are sharing the use of the same identifier. In ZEROIN, this occurs in two places, the identifiers TOL1 (in lines 011-012 and in the remainder of the program) and Q (on line 064 through the right hand side of line 068 and the remainder of the program, including the left hand side of line 068). Instances of this practice can be detected by data flow analysis, a techniques commonly used by compilers when optimizing the code it has generated.

3.6 Generalization of Interpolation Schemes

ZEROIN exhibits a situation where two sections of code use alternative approaches to compute the values of the same set of variable. Both lines 057-059 and 064-068 are responsible for computing 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*. Both computations and the test can be replaced conceptually by a more general expression that is responsible for computing 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 (encapsulating) the uses of the variables S and R inside of the new expression.

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

3.7 State Introduction

A common practice in empirical programming languages is to save the result of a computation in order to avoid having to recompute the same value at a later time. If the computation is involved, this practice can result in a significant savings at run time with a modest cost (one additional variable being introduced).

In ZEROIN, this practice has been used extensively. In particular, there has been a concerted effort to save the results of calls to the user-supplied function, F, in the variables FA, FB, FC. Because F may be arbitrarily complex, this practice may be the most important determinant of the ultimate efficiency of ZEROIN.

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 point of view of understanding the program, these three additional variables do not provide a significant abstraction. On the contrary, they require a non-trivial effort to understand and manipulate. Replacing them by their definitions makes the resulting program representation easier to understand.

There are other examples in the program of the introduction of a variable to capture the result of a computation. From the point of view of comprehension, there are two factors to be weighed in deciding whether to view the program with the variable name replaced by its value or left alone. On the one hand, each new variable places a burden on the person trying to understand the program. The variable must be read and its purpose understood and confirmed. On the positive side, variables can serve as valuable abbreviations for the computation that they replace. It is easier to understand a variable with a carefully chosen mnemonic 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. XM lays somewhere in the middle.

3.8 Other Small Examples of Representation Decisions

There are several additional instances where knowledge of the FORTRAN language and how to program in it are of value in understanding ZEROIN.

RETURN Statement: Functions in FORTRAN return their results by assigning them as the value of a special variable who's name is the same as the name of the function. In the case of this program, the variable ZEROIN is used only for this purpose.

Conditional Expression: Lines 092-093 represent a standard *conditional* statement. The value of the variable B is being adjusted by one of two factors based upon a test condition. Using a conditional expression instead of two IF statements avoids the duplication of the test.

Programming Idiom: Sign Test Line 095 represents a test to determine whether the values of the function F at the points B and C are the same.

Programming Idiom: Variable Swap A similar situation can be found on lines 032-034. This is an example of the classical idiom to swap the values of two variables. Once again a direct statement of this improves comprehension.

If-Then-Else: Lines 072-073 can be reformulated as an **if-then-else** statement. This requires an understanding of what absolute value means so it may be beyond the power of a simple program transformation system.

Programming Idiom: Computation of Machine Precision Lines 009-012 compute the value of EPS, a small floating point number used to determine whether the computed root is close enough to the actual root. It accomplishes this by successively dividing EPS by 2.0 until a result is computed that is too small to effect the result of an addition to 1.0. This value will be a constant for a particular machine and could have been supplied as a literal or through a call on an library function. In either case, the value could be referenced directly on line 041.

3.9 Generalization of Interval Computation

Thus far, the analysis of the program has proceeding without any knowledge of the problem domain, root finding. Because automated tools are unlikely to contain much knowledge of the wide variety of possible applications, it is desirable to isolate the problem elements into explicit chunks. An example of this was found in the generalization described in Section 3.6.

Now that the recognition of some intermediate design has clarified the structure of the program, the same sort of observation can be made about lines 048-086. They have the function of assigning values to the variable D and E based on the values of the variables A, B, C, D, E, F, TOL1, and XM. The fact that the list of variables is so long indicates that this segment is fundamentally interleaved with the rest of the program. Nevertheless, it is of value to indicate that the only explicit effect of these lines of code is to set these two variables.

It should also be noted, that as in Section 3.6, there are several instances of *specialization.* Lines 079-080 and 085-086 are selected based on the tests on lines 077-078. Likewise, the lines between 049 and 050 and lines between 050 and 087 are special cases selected on the basis of the tests on lines 048 and 049.

3.10 Program Architecture

Once the analysis described above has been performed, it is possible to appreciate the overall structure of the program. Based on the value returned on line 044, the program can be seen to use the variable B to hold approximations of the root of the function. B is modified on line 092 by either XM or D. The sections starting on line 095 and on line 031 act as adjustments that are made 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. The resulting architecture of the program is isolated in Figure 2.

```
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

3.11 Summary

Basili and Mills make extensive use of program proving techniques to understand and document ZEROIN. Although systems exist to assist in the verification of a proof, the automatic generation of loop invariants and the full automation of program proving is a long way off.

Computer Science research describes two approaches to reasoning about imperative programs: an assertional approach using invariants and, more recently, a transformational approach. While Basili and Mills used the assertional approach we have applied a transformational one.

The examples described in this section make use of the characterization of design decisions described in

Section 2. Recognition of instances of these activities does not require knowledge of the problem domain, but can use analysis techniques commonly found in compilers Most importantly, the separation of the examples emphasizes different aspects of the program structure that may not be apparent when a single analysis approach is used.

The assertional reasoning used in Basili and Mills finds its formal foundations in the the area of program verification. The transformational approach used in this paper is grounded in the newer techniques of transformational programming. The two approaches are complementary. Assertional reasoning can be used to identify design dependencies in imperative programs. The transformational approach is useful in that it permits the use of powerful transformations to restructure the program. By restructuring the program text, a programmer can simplify the task of recognizing important design dependencies.

4 THE FUTURE FOR DESIGN DECISIONS

Software maintenance require 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 structure the software together with details about the decisions which lead to that design.

Design decisions are not as yet fully understood. Indeed, many widely used notations for software design do not have the resolving power needed to represent and document design decisions. To understanding the design decisions underlying the implementation, a programmer must be able to identify high-level components, design dependencies, and transformations. In the future the programmer's efforts to identifying components, design dependencies and transformations should be automated, at least in part. Many tools for collecting and analyzing data from the program text — including cross-reference listings and diagrams tracing the flow of control and data within the program — can be based on technology previously developed for compilers.

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

References

[1] Victor R. Basili and Harlan D. Mills, “Understanding and Documenting Programs,” *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 3, May 1982.

[2] Martin S. Feather, “A Survey and Classification of some Program Transformation Approaches and Techniques,” *Program Specification and Transformation*, pp. 165-195, North Holland, 1987.

[3] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*, The MIT Electrical Engineering and Computer Science Series, McGraw-Hill Book Company, 1986.

[4] Lawrence J. Peters, *Software Design: Methods and Techniques*, Yourdon Press Computing Series, Prentice Hall, 1981.

[5] Eliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman and Robin Lamport, “Designing Documentation to Compensate for Delocalized Plans,” *Communications of the ACM*, Vol. 31, No. 11, November 1988.