RECOGNIZING DESIGN DECISIONS IN PROGRAMS

Richard J. LeBlanc, Jr.
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

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

1 DESIGN DECISIONS

During the course of development of a program, many decisions are made. Some relate to the problem domain and how it should be viewed and modeled. Others address constraints imposed by the solution space, including the target machine and programming language. Some decisions stand alone and have little impact 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. In order to effectively maintain an existing system, it is essential for a maintenance programmer to sustain previously made decisions unless the reasons for the decisions have also changed. In order to accomplish this the decisions must be recognized and understood.

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. Functional specifications may be formal or informal but are primarily concerned with what the target system is supposed to do and 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 such as target machine characteristics intrude. The middle ground between specifications and code is more nebulous. Webster[1] surveys the variety of notations and graphical representations that have been used. The design process as a whole can be described 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.

Design involves making choices among alternatives. Too often, however, the alternatives that are considered and the rationale for the final choice are lost. 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 attempt to represent the incremental changes that come with individual design decisions. Also, they 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 intent of the design are not retained with the code.

Design decisions are not made in isolation. Often a solution idea is best expressed through several interrelated decisions. Unless the interdependencies are explicitly documented, the unwary maintenance

programmer will fail to notice all of the implications of a proposed change. Design ideas that are expressed via interrelated decisions are called *delocalized information* by Balzer[2] and *delocalized plans* by Soloway.[3]

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

2 CHARACTERIZING DESIGN DECISIONS

Studying various areas Computer Science reveals several categories of design decisions. Abstraction mechanisms in programming languages provide evidence of the need to express design ideas in code. Semantic relationships from data base theory support the modeling of information structures from a variety of fields. Finally, examination of tools used for reverse engineering and software maintenance indicate decisions that have been found useful in understanding existing programs.

2.1 Composition and Decomposition

Probably the most common design decision made when developing a program is to split it into pieces. This can be done, 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 allowing decisions to be deferred and details hidden. Complexity is managed by using an appropriate name to stand for a collection of lower level details.

If a "top-down" approach is taken to design, then a program is *decomposed* into pieces. If a "bottom-up" approach is used, then a program is *composed* from available sub-components. 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 programming 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 of collecting 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.

2.2 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 the decision to gather selected parts of a program into a component, variously called a package, cluster, or module. The component's behavior is restricted by a protocol or interface so that other parts of the system can only interact with the component in limited ways. Parnas[4] introduced the term *information hiding* to describe this approach to structuring a system.

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 total amount of code contained within the component. If the component hides the details of a major design decision, then when that decision is altered during later maintenance, side-effects of the change are limited.

The alternative to 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 in a single loop. Interleaving in this way makes the resulting code harder to understand and modify. A number of useful interleaving transformations have been collected by Feather.[5]

2.3 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 procedures and functions are usually 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 software design.

Generalization is a design decision in which a program specification is satisfied by relaxing some of its constraints. For example, a program might be required to compute the logarithm of a limited set of numbers. The requirement could be satisfied by providing access to a general purpose library function for computing logarithms. The library function would be capable of computing logarithms of all of the set of required numbers as well as many others. The decision to use the library function is a generalization decision.

Abstractions other than numerical computations may also be parameterized. The Ada programming language provides a *generic* facility that allows data types and functions to parameterize packages and subprograms. Many languages provide macro capabilities that parameterize textual substitutions. Variant records in Pascal and Ada and type unions in C are examples 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 for 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 offer.

Specialization is a design decision related to generalization. Specialization involves replacing a program specification by a more restricted one. Often an algorithm can be optimized based on restrictions in the problem domain or facilities of the programming language. Although these optimizations can dramatically improve performance, they have a cost in lengthening the program text 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.

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

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. Generality has a cost, however. Generalized components may be less efficient than specially tuned versions. Moreover, there is often more

effort required to test a component intended for wide application than its more specific counterpart.

2.4 Representation

Representation is a powerful and comprehensive design decision. Representation is used when one abstraction or concept is better able to express a problem solution than another. This may arise 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, a programmer may choose a linked list to implement a pushdown stack. Bit vectors are used 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 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 and an index variable. This is a representation choice.

When the distinction between specialization and representation is kept in mind, representation can be seen to be a flexible and symmetric decision. In one context it may be appropriate to represent one construct by another. In a different situation, the inverse representation might be used. For example, operations on vectors are usually implemented 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.

2.5 Data and Procedure

Variables are not necessary in order to write programs; values can always be explicitly 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.

Programmers must be aware of the invariants relating the program variables when inserting statements

into a program. For example, suppose a maintenance programmer is investigating 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 parts of the program, 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 replace references to variables with the computations that produced their most recent values. The resulting statements can be rearranged 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. 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 ordering constraint or when variables are computed in the wrong order.

The alternative to introducing a variable is to recompute values when they are needed. This is sometimes used to make a program more readable. 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 associated with 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 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, the next-state information can be computed directly in code for each of the states. Although this may seem unusual, it is exactly the technique that is 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 programming language. The reason for doing this is efficiency: in the procedural version the cost of indexing into the array is avoided.

2.6 Function and 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, instead, 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 be left up to the user at run-time instead of the developer at design time.

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

Of course, more traditional programming languages do not support non-deterministic 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; that is, 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 of value to provide the inverse function that, when given a percentage, returns the numeric grade that would separate that proportion of the students.

3 FINDING DESIGN DECISIONS IN CODE

Software maintenance and reuse activities require the detection of design decisions in existing code, which is a part of 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. The process is bottom-up and incremental; low level constructs are detected and replaced by their high-level counterparts. If this process is repeated, gradually the overall architecture of the program emerges from the programming language-dependent details.

The program below is taken from a paper by Basili and Mills[6] in which they use flow analysis and techniques from program proving to guide the comprehension process and document the results. It will be used as a realistic example of production software in which design decisions can be recognized. The program is shown in Figure 1.

```
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 RELATIVE MACHINE 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 BY ZEROIN 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

ZEROIN 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.

3.1 Interleaving of 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 is used for no other purpose.

This leads to the conclusion that the overall program can be decomposed into two pieces, the root finder and the debugging printout. To make the analysis of the rest of the program simpler, the diagnostic portion can be removed from the text being considered. 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 Representation of Structured Control Flow in Fortran

Basili and Mills begin their analysis by examining the control flow of the program. 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 developer. 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 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 control flow structure of the program at the expense of complicating 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 two properly formed conditional constructs.

3.4 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 share the use of the same identifier. This occurs twice in ZEROIN, with 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.

3.5 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 variables. 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 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 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 allows the details of the individual cases to be hidden away. This, in turn, makes the code shorter and easier to understand.

3.6 Variable Introduction

A common programming practice 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.

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, and 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 algorithm, 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 easier to understand.

When readability is the goal, there are two factors to be weighed in deciding whether to write the program using a variable name or replacing it by its value. 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, albeit without the benefits of mnemonic names. XM lies somewhere in the middle.

3.7 Generalization of Interval Computation

Now that the recognition of some intermediate decisions 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 highly 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 3.5, there are several instances of *specialization*. Lines 079-080 and 085-086 are selected based on the tests on lines 077-078. Likewise, lines 082 through 086 and the lines between 050 and 081 are special cases selected on the basis of the tests on lines 048 and 049.

3.8 Program Architecture

Once the analysis described above has been performed, it is possible to appreciate the overall structure of the program. Based on the test made 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 lines 092 and 093 by either XM or D. The sections on lines 025-028 and lines 031-037 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 shown 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

4 REPRESENTING DESIGN DECISIONS

It is not sufficient to simply recognize design decisions in code. Once recognized, the decisions must be organized in such a way that they can be effectively used by maintenance programmers and reuse engineers. The organization chosen serves as a representation for design information.

There are numerous methods for designing software and numerous representations for the intermediate results. 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.

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 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.

Several approaches to organizing design information have been proposed. Biggerstaff[7] is concerned with relating code fragments to information from the problem domain. Software reuse will be facilitated if a new problem's requirements can be easily matched against a description of existing software. He is building the Desire system to explore his approach. Blackburn[8] 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). Coleman and Gallimore report on FPD, a framework for program development.[9] 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.

5 CONCLUSION

Software 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 structure of the software together with details about the decisions which lead to that structure.

Design decisions occur where the 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 greater hope of being able to maintain and reuse the mountains of undocumented software confronting us.

References

1. Dallas E. Webster, "Mapping the Design Representation Terrain: A Survey," MCC Technical Report Number STP-093-87, July 1987.

2. Robert Balzer, "A 15 Year Perspective on Automatic Programming," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, November 1985, pp. 1257-1268.

3. 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.

4. D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol. 15, No. 12, December 1972, pp. 1053-1058.

5. Martin S. Feather, "A Survey and Classification of Some Program Transformation Approaches and Techniques," *Program Specification and Transformation*, pp. 165-195, North Holland, 1987.

6. Victor R. Basili and Harlan D. Mills, "Understanding and Documenting Programs," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 3, May 1982.

7. Ted J. Biggerstaff, "Design Recovery for Maintenance and Reuse," *IEEE Computer*, July 1989.

8. Mark R. Blackburn, "Toward a Theory of Software Reuse Based on Formal Methods," Software Productivity Consortium Technical Report, SPC-TR-88-010, Version 1.0, April 1988.

9. Derek Coleman and Robin M. Gallimore, "A Framework for Program Development," *Hewlett-Packard Journal*, Vol. 38, October 1987, pp. 37-40.