On the Knowledge Required to Understand a Program

Richard Clayton, Spencer Rugaber, and Linda Wills
Georgia Institute of Technology

Abstract

This paper is concerned with the units of knowledge used in understanding programs. A pilot study was conducted wherein a short, but complex, program was examined looking for "knowledge atoms," the units from which program understanding is built. The resulting atoms were categorized along three orthogonal axes of knowledge type, design decision used, and the type of analysis required to uncover the atom. The results are discussed relative to several approaches to program understanding taken from the research literature.

Keywords: program understanding, reverse engineering

1. Motivation

The ultimate goal of research in program understanding is to improve the process of comprehending programs, whether by improving documentation, designing better programming languages, or building automated support tools. This laudable goal is inherently unattainable, however, for the following reason. We have no agreed-upon definition or test of understanding! The closest measure of which we are aware is Shneiderman's 1970s experiments in software psychology that gauge understanding by the extent to which subjects can recreate a program from memory [22]. While this measure surely qualifies, it is naturally limited to small programs.

Researchers in natural language understanding have a similar difficulty. However, early on, Alan Turing proposed one way out of the problem, his now-named *Turing Test,* by which human judges could communicate using electronic means with the inhabitants of two closed rooms, one containing a computer and one containing a person [25]. If the judges, by various trial communications, could do no better than chance at determining which room held which occupant, the computer would be deemed to have obtained human-level abilities to understand and communicate.

The field of program understanding has no such test. In fact, there is no real agreement on what it is about a program that needs to be understood. For example, Shneiderman talks about *syntactic* and *semantic* knowledge [23]. Pennington uses the terms *domain world* and *programming world* [13]. It is the purpose of this paper to explore this issue. That is, to propose a characterization of the units of knowledge from which program understanding can be built.

To explore this question, the authors examined a nontrivial program looking for knowledge atoms, individual facts, the sum total of which would constitute complete understanding. The atoms were then categorized along several dimensions looking for overlap, inconsistency, and gaps. Although not conducted as a rigorous experiment, the investigation did produce sufficient insight that a theory of program knowledge atoms can be proposed and confirmatory experiments designed.

2. Procedure

We examined the **ZEROIN** program, which finds a root of a function of a single, real variable in an interval [4]. The program is widely used, and, despite containing only 102 lines of FORTRAN code, is complex, providing significant challenges to understanding, even by experienced programmers.

The analysis procedure assumes a reader knowledgeable in Fortran, reading for understanding rather than to perform a specific maintenance task such as fixing a bug. The procedure also assumes that the reader actively engages the text by raising relevant questions while reading and attempts to answer the questions with the material read so far. Analysis via active reading has the advantage of being well known and frequently recommended understanding technique [2]. Further, the analysis procedure is largely independent of the various definitions of program understanding available: which facts are relevant and how relevant facts are used may vary from one form of program understanding to another, but how the facts are gathered is the same.

The reading assignment was to obtain an overall understanding of **ZEROIN**. While reading, each fact used to answer questions posed during the reading is recorded, along with an indication of the program statements to which the fact applied, and, for facts derived from the program, the statements from which the fact arose. Facts related exclusively to understanding the program's FORTRAN syntax are explicitly excluded.

After completing the reading, each recorded fact is categorized as a knowledge atom in three orthogonal ways: by knowledge category, by design-decision category, and by automated-analysis category.

Knowledge category: place the atom in any of the following traditional categories.

- Domain knowledge - knowledge of numerical analysis and root finding;
- FORTRAN language knowledge - non-syntactic knowledge of language semantics and run-time library behavior;
- Programming knowledge - language-independent knowledge of algorithms, data structures, and idioms.

Design decision category: place the atom according to the type of design abstraction it expresses. In previous work [19], one of the authors and his colleagues developed a categorization of programming knowledge based upon the type of design decision used to develop the program. The particular categories are discernible by the use of various abstraction mechanisms provided by the programming language. For the pilot experiment, each of the detected knowledge atoms were placed into any of the following six categories.

- Decomposition - a decision to break a concept into pieces;
- Specialization - a decision to treat a situation in terms of one of several special cases of a general concept;
- Dispersal - a decision on how best to organize the elements of an abstraction, either by encapsulating them or interleaving them with the elements of another abstraction;
- Representation - a decision on the use of one mechanism to implement another;
- Operationality - a decision on how declaratively to express a concept;
- Modality - a decision on whether an implementation is more relational or functional in nature.

Further explication of these categories is given in Section 5.2.

Automated analysis category: place the atom according to the type of automated analysis required to detect it. One of the authors, in a previous article has described the following categories of automated analyses [18].

- Textual - measuring simple properties of the source code text, e.g., the number of source lines of code predicts the amount of effort required to comprehend a program;
- Lexical - decomposing the sequence of characters in a program's source listing into its constituent lexical units (e.g., identifiers, operators, keywords, strings, and numbers). This enables the generation of cross-reference listings and complexity metrics based on the relative frequency and uniqueness of identifiers and operators;
- Syntactic - parsing source text into a hierarchical organization of its syntactic structure, based on constituents such as expressions, statements, and modules defined by the grammar of the source code's programming language;
- Control flow - analysis to determine the order in which statements can be executed within a subprogram and the calling relationships among subprograms;
- Data flow - analysis of the dependencies between definitions and uses (or references) of variables;
- Semantic - derivation of semantic properties of a program through mathematical techniques based on denotational semantics, such as abstract interpretation.

Section 3 presents several examples of the knowledge atoms categorized. After categorization, the resultant data was stored into a relational database management system, on which queries could be written and analyses performed. The results of these analyses are presented in Section 4. Previous, alternative approaches to analyzing the **ZEROIN** program are described in Section 5.

3. Examples

This section presents some examples of the knowledge atoms obtained when reading **ZEROIN**. The complete analysis can be found in reference [17]. The intent of the pilot study was to simulate the following situation. You are assigned responsibility for maintaining a program you have never seen before. It is written in the FORTRAN language and is concerned with finding the roots of a function. We will assume that you know the FORTRAN language but are not an expert at it. That is, you still have to occasionally look at the reference manual to answer questions about the language. We will also assume that you have a computer science background, either through formal education or by on-the-job experience, so that you know how to design and compose programs similar to the one you are about to maintain. Finally, we will assume that you have a passing acquaintance with numerical analysis, possibly from a course you took. Hence, you are familiar with the idea of finding a root of a function, but you would have to look up an actual algorithm in order to write a root-finding program yourself.

As you are responsible for long-term maintenance of the program, you want to understand it better. So you decide to systematically read it. This is distinct from the situation where you have a specific task to accomplish, such as finding a bug, adding a new feature, or updating the program to conform to a change in the language or operating environment. In these cases, instead of systematic reading, you might direct your efforts to accomplishing the specific task. Here, we will assume that you are going to make a single, sequential pass through the program text, with perhaps a few side trips to answer small questions as they arise.

The program text found in the Appendix is taken directly from its source [4]. We have added three-digit line numbers in the left margin for expository purposes; they are not part of the program itself. The examples selected are intended to represent the different kinds of knowledge required to understand the program. In some cases, of course, a single pass is insufficient to gain understanding, and the best that can be expected during reading is to raise questions for later resolution. We have presented examples of such questions in italics.

Example 1: Computation of Relative Machine Precision

```
  9         EPS = 1.0
 10      10 EPS = EPS/2.0
 11         TOL1 = 1.0 + EPS
 12         IF (TOL1 .GT. 1.0) GO TO 10
```

Understanding these lines of code requires knowledge of floating point arithmetic. Without this knowledge, it looks like we keep dividing **EPS** by **2.0** until we cannot distinguish it from **0.0**! If this is the case, then the whole block of statements reduces to a no-op. In fact, the purpose of the block is to find a very small floating point number, but one that acts like **0.0** when it is added to **1.0**. Without reading the rest of the program, we cannot tell why this information is needed, consequently, we are left with questions for later resolution: Are we trying to compute a value for **TOL1** or for **EPS**? Which of these are referred to later in the routine, and which are temporary variables?

Several other issues arise when considering these statements. One concerns the use of labels and the **GO TO** statement. Whenever we come across a labeled statement, several questions come immediately to mind. *Is it used for flow of control or is it a **FORMAT** statement? If the former, what statements refer to (can branch to) this label? Under what circumstances can those branches be taken? That is, what can we assume is true after such a branch has been taken?*

Lines **10-12** actually implement a **do-while** loop such as is found in the C language. That is, these FORTRAN lines provide the programmer a *representation* for a nonexistent language feature. Making an imaginary transformation, we get the following replacement statements.

```
 9          EPS = 1.0
            DO
10            EPS = EPS / 2.0
11            TOL1 = 1.0 + EPS
12          WHILE (TOL1 .GT. 1.0)
```

Another issue concerning this block arises every time we have a loop. *Are we even sure that the loop terminates?* Might there not be a "smallest floating point number" which when divided by two and the results rounded upward, yields itself. Only a thorough understanding of floating point arithmetic and the hardware of the machine on which the program is run can answer the question.

The code in **9-12** makes no reference to the input parameters. This raises another issue, *should the value of **EPS** be computed externally to **ZEROIN** and passed in as a parameter to avoid repeated recomputation? Or should it be computed at compile time and available in some form of environmental constant such as is provided in the ANSI C language?*

Example 2: Interleaving

```
16        IF (IP .EQ. 1) WRITE (6,11)
17     11 FORMAT('THE INTERVALS DETER
              MINED BY ZEROIN ARE')
```

IP is an input parameter. In fact, its only use in **ZEROIN** is to control debugging printout. Those lines in the program that refer to it (**16-17** and **29-30**) could be removed from the program without affecting the roots that it computes. It is as if **ZEROIN** is trying to accomplish two things simultaneously: root computation and debugging printout. This is an example of interleaving [20]. To assure ourselves that this is the case, however, we must answer several questions about the function. *Is **IP**'s value ever set in **ZEROIN**? Or can it be treated as a read-only parameter? What other values can **IP** hold? Is it being used as a flag? Why does the program perform output inside of a function?*

Example 3: State introduction

```
18        A = AX
19        B = BX
20        FA = F(A)
21        FB = F(B)
```

Two programming tricks are illustrated by these four lines. In the first two lines, new state is introduced: copies of input parameters are being saved into local variables so that the copies might be subsequently altered without corrupting the original values. Understanding this fact requires knowledge of how FORTRAN passes parameters and answers to the following two questions. *Are there any other references to **AX** and **BX**? Are the values of **AX** and **BX** ever changed?*

The second programming trick is illustrated by lines **20-21**. **F** is an example of a parameter to a function that is itself a function. **F** is actually invoked in lines **20-21**. The name ***FA*** suggests that it serves as a surrogate for the call to **F** with parameter **A**, perhaps introduced to save what might be a costly re-execution. In fact, we can hypothesize an invariant condition: **FA** shadows the value of **F(A)** and whenever **A** changes, **F(A)** should be called and assigned to **FA**. To check this requires scanning the entire program to see if there is ever any violation.

Example 4: Variable name reuse

```
41     40 TOL1 = 2.0*EPS*ABS(B) + 0.5*TOL
```

The **TOL1** variable name is being recycled here. We can think of the memory location named by **TOL1** as being interleaved between its uses in lines **11-12** and its uses in the rest of **ZEROIN**. *How can we confirm this?* We have to make sure that no path from the previous definition can reach any reference to **TOL1** without going through the current statement. Data flow analysis of **ZEROIN** is required to confirm this.

Example 5: Interval midpoints and convergence

```
42          XM = .5*(C-B)
43          IF (ABS(XM) .LE. TOL1) GO TO 90
44          IF (FB .EQ. 0.0) GO TO 90
```

Understanding these three lines of code requires knowing something about root finding—that it works by shrinking an interval containing a root until the interval length passes a convergence test. In particular, if **C** and **B** define the current interval, then **C-B** is the interval length, and **XM** is the distance to the midpoint. *Could the name* **XM** *stand for the* **X** *value of the distance to the **M**idpoint of the interval?*

Lines **43-44** is a candidate for the termination test of the convergence loop. *Is line **90** the function exit?* The two values being compared correspond to one half of the length of the interval (**XM**) and the size of the region (**TOL1**). When the interval gets small enough **(.LE. TOL1)** then it doesn't make sense to continue shrinking it. If our hypothesis is correct about **FB** being always the value of **F(B)**, then another reason for quitting is that **FB** is **0.0**; that is, that we have found a root.

Example 6: Alternative interval shrinkage methods

```
57          S = FB/FA
58          P = 2.0*XM*S
59          Q = 1.0 - S

64     50 Q = FA/FC
65          R = FB/FC
66          S = FB/FA
67          P = S*(2.0*XM*Q*(Q-R)-(B-A) *
               (R-1.0))
68          Q = (Q-1.0)*(R-1.0)*(S-1.0)
```

Lines segments **57-59** and **64-68** have similar appearances and roles. They both compute some intermediate values prerequisite to shrinking the interval boundaries. In fact, the two segments are both special instances of the general desire to find an intermediate point in the interval to serve as a new interval boundary. In order to assure ourselves of this, however, requires answering several questions. *Are* **S, P,** *and* **Q** *used similarly outside of these segments? Is the value of* **R** *computed in line* **65** *used anywhere outside of these lines?*

Example 7: Programming style

```
77        IF ((2.0*P) .GE. (3.0*XM*Q -
            ABS(TOL1*Q))) GO TO 70
78        IF (P .GE. ABS(0.5*E*Q)) GOTO 70
```

Here are two adjacent statements both making tests whose consequent is to skip the remainder of this section of **ZEROIN**. As there are no side effects present, *why not just combine these two lines into one with an* **.AND.***?* One possible reason is to guarantee order of evaluation; that is, to make sure that the first condition gets checked before the second. If **.AND.** is used instead and if the language definition allows, some compilers might take the liberty to reorder the evaluations of these two conditions. This is a language semantics question. Another possible reason is that both of these two conditions have their own relevance to the interpretation of the underlying algorithm. That is, it makes the program more readable to see the two conditions on separate lines.

4. Results

4.1 Raw data

ZEROIN comprises 102 lines of Fortran. Of these, 41 are comments, 28 of which are, in turn, blank; that is, provided only to improve the program's readability. Of the remaining 61 lines, four are not executable, being either declarations or indicating the end of the function. Of the 57 executable lines, 36 are assignments, 15 are **IF** statements, two are **FORMAT** statements, three are **GO TO** statements, and one is the **RETURN** statement. For the 36 assignment statements, the right hand sides of seventeen of them has no operations, fourteen have one, two have two, two have five, and one has nine operations. These observations roughly follow the statistics measured by Knuth in his empirical study of Fortran programs [8].

While reading the 102 lines, 71 knowledge atoms were detected. The range of program statements over which an atom applies gives a rough estimate of the atom's complexity; the more statements over which an atom ranges, the more complex it is. Within the context of a single, sequential reading of the source code, 54 atoms ranged over a single statement, one ranged over two statements, fifteen ranged over three statements, and one ranged over four statements.

4.2 Traditional categorization

The traditional breakdown of the knowledge required to understand a program distinguishes between syntactic and semantics knowledge [23]. The semantic category, includes knowledge of the programming language's semantics, of the application domain, and of programming itself. The results obtained in the pilot study are presented in Table 1, where **F** denotes knowledge of Fortran, **D** denotes knowledge of the domain of root finding, and **P** denotes knowledge of programming.

Table 1: Traditional Knowledge Sources

Knowledge Source	Count
F	8
P	0
D	6
F & **D**	42
P & **D**	1
F & **P**	1
F, P, &D	8
None	5
Total **F**	59
Total **P**	10
Total **D**	57

Lines in the *None* category correspond to parts of **ZEROIN** whose purpose could not be determined based on a single, sequential reading of the program.

Discussion: The collected data indicate strong overlap (or inability to distinguish) between Fortran and domain knowledge. This is not surprising when considering that Fortran is being used to solve a domain problem. Moreover, the fact that the program is short and involves little in the way of data structures, suggests why the programming category is as small as it is. Overall, the traditional breakdown has not given us much guidance on either how to organize the knowledge required to understand the program or on how to structure tools to support understanding.

4.3 Design decision categorization

Several different types of design decisions were detected in reading **ZEROIN**, located primarily in 23 knowledge atoms. The dominant decision type was choice of representation, which participated in 14 (or 61%) of the 23 atoms involving a decision. A related decision that overlapped with 11 of the representation decisions is the operationality decision: i.e., whether to use a declarative data representation (chosen in 15 atoms) as opposed to a procedural mechanism (chosen once) to manage some computation. **ZEROIN** also featured one decomposition decision and three instances of each of the specialization vs. generalization and the dispersal (i.e., encapsulate vs. interleave) decisions. At least one knowledge atom was associated with each of the categories of design decisions except modality. While only 23 knowledge atoms involved making a design decision, almost all those that did involved more than one interacting decision.

Discussion: Design decisions are harder to characterize and understand than the traditional knowledge categories. This might explain the reduced number of atoms detected. The high incidence of representation decisions is characteristic of programs written in older languages where the programmer is forced to simulate modern structured control structures.

4.4 Type of analysis

Ultimately, of course, it is desirable to automate as much as possible of the software understanding process. There currently exist a variety of analysis techniques that are routinely applied, and many more have been proposed. The question is, which ones are really useful in supporting understanding? Of those routinely available, Table 2 shows how frequently they occurred when the knowledge atoms from **ZEROIN** were considered.

Table 2: Type of Analysis

Type	Count
Textual	15
Lexical	34
Syntactic	56
Control Flow	36
Data Flow	23
Semantic	53

Discussion: It should come as no surprise that syntactic analysis is frequently required to understand many knowledge atoms. The prominence of abstract syntax trees as an intermediate representation mirrors this observation. The semantic category is also strongly represented, probably because it serves as a way to categorize all forms of analysis more powerful than dataflow analysis.

4.5 Overall discussion

Because of the informal and preliminary nature of our study, we have not attempted to compute cross-correlations among the categories. Nevertheless, such statistics would provide insight into questions such as which kinds of analyses are required to detect domain knowledge, what is the relationship between programming knowledge and design decision categories, and what kinds of semantic analyses are really required to answer programmers questions about a program?

5. Alternative approaches

A single-pass, top-down approach is not the only way to read a program. One alternative is to examine the program bottom-up, building up a description of overall program functionality from the computations performed on the specific lines. This can even be done formally, with a mathematical description being given to the computations. Basili and Mills have done this for the **ZEROIN** program [3], and we summarize their approach in the next subsection.

Another possibility is to combine a top-down and a bottom-up approach. The bottom-up approach looks at the mechanics of each statement from the point of view of design decisions made by the programmer while a synchronized, top-down process is building up a description of the problem the program is solving. Rugaber, Ornburn, and LeBlanc have described this approach for **ZEROIN** [19], and we summarize it in the second subsection to follow.

Our empirical study of the knowledge atoms underlying program understanding is also related to research in using programming plans and design patterns as a basis for program understanding. We discuss this relationship in the remainder of the section.

5.1 Basili and Mills

Basili and Mills apply ideas from structured programming and program correctness proofs to the understanding and annotation of computer programs [3]. In particular, they construct a *prime program decomposition* of the program, define *program functions* for each prime program, build a data reference table describing the uses of each program variable, and then synthesize a program correctness proof demonstrating exactly how the program accomplishes its goals.

A *proper program* is a contiguous program segment with a single entry point, a single exit point, and the property that each statement in the segment is on a path from the entry to the exit; that is, that each statement is actually used during execution. A *prime program* is a proper program that does not contain any more-basic proper program except for the individual program statements. Roughly speaking, a prime program corresponds to a structured control construct. In fact, for the two prime programs in **ZEROIN** that contained more than one predicate, Basili and Mills manually replaced them with structured control constructs.

A useful property of prime programs is that they can be composed. That is, larger program units can be built out of smaller ones while still retaining the single entry, single exit, composition property. Thus, we can compose the meanings of the prime units in the same ways that we can compose them syntactically.

Program functions are predicates that describe how a statement or larger program segment produces its output values in terms of its inputs. They are abstractions that summarize what a series of statements accomplishes without going into the details of how they accomplish it.

A *data reference table* is like a cross-reference listing that breaks out references into those that set or define a variable from those that merely access or use it.

Using their approach, the authors are able to gain an understanding of **ZEROIN**. The approach helps them organize their analysis and provides a completeness criterion. That is, they know that they have to keep working until they complete their proof. Hence, it gives them a way of knowing when their understanding is deep enough. It should be noted, however, that like other uses of program correctness proofs, the authors' approach to program understanding requires mathematical sophistication and a certain inventiveness for constructing loop invariant conditions and for recognizing the *indeterminate bounded variable* that they used to summarize the program's progress.

5.2 Rugaber, Ornburn and LeBlanc

Rugaber, Ornburn, and LeBlanc view programs as the results of *design decisions* made by a programmer during the course of development [19]. Understanding an existing program, therefore, requires recognizing and annotating its decisions based on the evidence provided by the source code.

The paper characterizes design decisions based on the kinds of abstractions provided by programming languages, by database modeling techniques, and by transformational programming theory. The decisions belong to one of several categories. A *composition* decision groups lower level constructs and gives them a single name. For example, variables can be grouped into a record structure, or program statements can be grouped into a subprogram. In **ZEROIN**, the programmer decomposed the algorithm into a series of paragraphs each introduced by a comment.

A separate but related decision is whether the internals of the resulting aggregate are visible to its clients. If not, the aggregate is said to be *encapsulated*. In the case of program statements, encapsulation can be enforced by the use of modules, abstract data types, or information hiding. An alternative to encapsulation is *interleaving* where the internals of two or more constructs are intentionally intermixed, usually for reasons of improved efficiency. Perhaps the most difficult aspect of understanding **ZEROIN** is appreciating how it interleaves three different methods for shrinking the interval during each iteration of the main loop.

The three methods for shrinking the loop also illustrate a *specialization* design decision. That is, the three ways of shrinking the interval all have the effect of updating the program state for the next iteration. Specialization is even easier to accomplish in object-oriented programming languages where inheritance can be used to quickly define functionality in a subclass that specializes that in its superclass.

A fourth category of design decision is called *representation*. Representation is typically seen when one type of data structure, such as an array, is used to implement another, such as a stack. In **ZEROIN**, an interval is represented by a pair of floating point numbers. Additionally, the programmers who wrote **ZEROIN** made disciplined use of its **GO TO** statement to represent more structured, but unavailable, control structures, such as the **if-then-else** statement.

Programmers often have to decide whether to compute a value or look it up. In ZEROIN, for example, the variables **FA**, **FB**, and **FC** are used to hold the values of **F(A)**, **F(B)**, and **F(C)**, respectively. These values could have been computed when they were needed, but the programmer decided that a call to **F** might be costly, and looking up the value in a variable could accomplish the same purpose, even if it made the code marginally more difficult to understand.

The final category of design decision described by Rugaber, Ornburn, and LeBlanc has to do with functions and relations. **ZEROIN** is a FORTRAN function that takes as input another function, a tolerance, and an interval and produces a root. It is possible to imagine a related function that takes the function, tolerance, and root, and produces the interval. In fact, the Prolog programming language supports this duality by allowing, in many cases, the same code to be used for computing in both directions. That is, the Prolog program allows the designer to express the relation between the values (interval, tolerance, root, and function) without requiring specification of which values are inputs and which are outputs. While FORTRAN does not permit this degree of generality, specifications are often couched in terms of relations that must be explicitly implemented as functions.

After describing the categories of design decisions, Rugaber, Ornburn, and LeBlanc present examples of them in the code for **ZEROIN**. They suggest that the annotation of decisions such as those made during program development can significantly ease the burden of subsequently maintaining the program.

5.3 Programming plans

For several years, researchers have investigated the key role that plans play in understanding how the goals of a program are implemented in its code. A *plan* denotes a description or representation of a computational structure that the designers have proposed as a way of achieving some purpose or goal in a program [9][15][24]. Note that a plan is not necessarily stereotypical or used repeatedly; it may be novel or idiosyncratic, as is the idiom shown in Example 8.

Several techniques have been developed for automating the recognition of standard, stereotypical plans (or *clichés* [15]), such as [1][6][7][10][11][12][14][16][26]. Experiments with these recognition systems have shown the benefits of uncovering plans in programs to form a basis for understanding the programs.

Our categorization of knowledge atoms encompasses plans in addition to other forms of knowledge that are valuable in understanding programs. Our categorization describes plans at a fine level of granularity, cutting across multiple dimensions. For example, plans may be programming-knowledge-specific (such as standard ways of controlling a search) or domain-specific (such as a quadratic interpolation algorithm or interval shrinkage methods). Some are more amenable to detection based on control-flow analysis while others are manifested primarily in dataflow constraints. In addition, our categorization of knowledge atoms includes information about the design decisions or rationale underlying each unit.

While knowledge atoms capture the basic units of knowledge underlying a plan, plans impose causal structure on collections of knowledge atoms. They capture knowledge about the purpose of each atom in relation to the other atoms and the overall goals of the program.

In [6], Hartman proposed an empirical study of programs to determine their "planfulness" with respect to a given plan library. The *planfulness* of a program refers to the extent to which plans in the library and variations of them occur in the program. Such a study would be valuable in helping to confirm the hypothesis that recognizing plans is beneficial to program understanding and would also help tailor recognition techniques to a given program population and plan library. The study we have performed is an initial step toward an empirical study similar to and extending the one Hartman proposed. Realizing that plans are one form of knowledge that is useful in understanding programs, we would like to broaden the study of natural program populations to focus on the range of knowledge units we've identified. This will help validate and refine our categorization as a basis of the knowledge underlying program understanding.

5.4 Design patterns

Recently, researchers have started collecting, documenting and cataloging software design patterns [5][21]. These are recurring solutions to common problems in a context. They originally grew out of work on high-level design solutions in object-oriented programming and now encompass a variety of areas, including non-object oriented design, maintenance, and testing, and a range of solutions that feature nonfunctional properties and social and human factors.

Patterns are at a much higher level of granularity than knowledge atoms. In fact, knowledge atoms, plans, and patterns each form successive layers of an overlapping hierarchy with increasing granularity. Patterns are "molecular" in that they include knowledge from several knowledge atoms. Several different aspects of a recurring solution need to be present to be collectively recognizable as a pattern, including the pattern's problem context, consequent benefits, and drawbacks. They also include information about related patterns and collaborations with other patterns that are typically used in conjunction with them.

6. Conclusions and future work

This paper argues that we need to better understand the units of knowledge from which program understanding is constructed. Without such an understanding it will be difficult for us to make much progress toward our ultimate goal of helping practical programmers do their jobs better.

Understanding even a relatively small program is a complex process. It requires both knowledge and analysis. Knowledge is required of the programming language syntax, semantics, and run-time libraries, of the computing machine and its operating environment, of the application domain, including how problems in the domain are typically solved, and of programming in general—how programs are structured, variables used, and correctness guaranteed.

Much work is required to develop the ideas we have presented. We need to sharpen the definition of knowledge atoms and perform repeatable experiments to see whether programmers find the same units. We also need to sharpen our categories and compare the results obtained by different subjects when attempting to characterize a given atom.

We believe that properly organized knowledge is an essential ingredient of any automated understanding tool. But how should the knowledge be organized? Our answer is that it should be organized in the same units that people use when understanding programs manually. Consequently we have to better understand those units.

Acknowledgments

Effort sponsored by the National Science Foundation (CCR-9708913) and the Defense Advanced Research Projects Agency, and the United States Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-2-0229.

References

[1] S. Abd-El-Hafiz and V. Basili. "A Knowledge-based Approach to the Analysis of Loops." *IEEE Transactions on Software Engineering,* 22(5):339-360, May 1996.

[2] Mortimer J. Adler and Charles Van Doren. *How to Read a Book.* Simon and Schuster, 1940.

[3] Victor R. Basili and Harlan D. Mills. "Understanding and Documenting Programs." *IEEE Transactions on Software Engineering*, SE-8(3):270-283, May, 1982.

[4] G. Forsythe, M. Malcolm, and M. Moler. *Computer Methods for Mathematical Computations.* Englewood Cliffs, New Jersey: Prentice-Hall, 1977.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Abstraction and Reuse of Object-Oriented Software,* Addison-Wesley, 1995.

[6] J. Hartman. *Automatic Control Understanding for Natural Programs*. TR AI 91-161, Ph.D. thesis, University of Texas at Austin, 1991.

[7] W. L. Johnson. *Intention-based Diagnosis of Novice Programming Errors*. Morgan-Kaufmann, 1986.

[8] Donald E. Knuth. "An Empirical Study of FORTRAN Programs." *Software — Practice and Experience*, 1(2):105–133, February, 1971.

[9] S. Letovsky and E. Soloway. "Delocalized Plans and Program Comprehension." *IEEE Software,* 3(3):41-49, May 1986.

[10] S. Letovsky. *Plan Analysis of Programs.* Research Report 662, Ph.D. thesis, Yale University, 1988.

[11] V. Kozaczynski and J.Q. Ning. "Automated Program Understanding by Concept Recognition." *Automated Software Engineering,* 1(1):61-78, March 1994.

[12] W. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems.* Morgan-Kaufmann, 1988.

[13] Nancy Pennington. “Comprehension Strategies in Programming.” *Empirical Studies of Programmers: Second Workshop,* G. Olson, S. Sheppard, and E. Soloway, editors, Ablex, 100-113, 1987.

[14] A. Quilici. “Memory-based Approach to Recognizing Programming Plans.” *Communications of the ACM,* 37(5):84-93, May 1994.

[15] C. Rich and R. C. Waters. *The Programmer’s Apprentice*. Addison-Wesley, 1990.

[16] C. Rich and L. M. Wills. “Recognizing a Program's Design: A Graph-Parsing Approach.” *IEEE Software*, 7(1):82-89, January 1990.

[17] Spencer Rugaber. “An Example of Program Understanding.” GIT-CC-98-14, May 1998.

[18] Spencer Rugaber. “Program Understanding.” Allen Kent and James G. Williams, editors, *Encyclopedia of Computer Science and Technology,* Marcel Dekker, Volume 35, Supplement 20, 341-368, 1996.

[19] Spencer Rugaber, Stephen B. Ornburn, and Richard J. LeBlanc, Jr. “Recognizing Design Decisions in Programs.” *IEEE Software*, 7(1):46-54, January, 1990.

[20] S. Rugaber, K. Stirewalt, and L. M. Wills. “Understanding Interleaved Code.” *Automated Software Engineering,* 3(1-2):47-76, June 1996.

[21] D. Schmidt, M. Fayad, and R. Johnson (eds.), Special issue on Software Patterns, *Communications of the ACM*, 39(10), October 1996.

[22] Ben Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems.* Little Brown, 1980.

[23] Ben Shneiderman and R. Mayer. “Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results.” *International Journal of Computer and Information Sciences,* (7)3, 1979, 219-239.

[24] E. Soloway and K. Ehrlich. “Empirical Studies of Programming Knowledge.” *IEEE Transactions on Software Engineering,* 10(5):595-609, September 1984.

[25] Alan Turing. Computing Machinery and Intelligence.” *Computers and Thought.* E. Feigenbaum and J. Feldman, editors. McGraw-Hill. 1-35, 1963.

[26] L. M. Wills. “Using Attributed Flow Graph Parsing to Recognize Clichés in Programs.” *Graph Grammars and Their Application to Computer Science*, J. Cuny, H. Ehrig, G. Engels, G. Rozenberg, editors, *Lecture Notes in Computer Science* 1073, pp. 170-184, Springer Verlag, 1996.

Appendix: ZEROIN

```
      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
```