Understanding someone else's code: Analysis of experiences

Arun Lakhotia

The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
(318) 231-6766, -5791 (Fax)
arun@cacs.usl.edu

To appear in the *Journal of Systems and Software*, 1994.
Earlier version appeared in the *Reverse Engineering Newsletter*, January 1993, a publication of the Subcommittee on Reverse Engineering of the IEEE Computer Society. Also available as USL technical report CACS-TR-92-5-10, November 1992.

Do we need to understand the whole code?

A friend of mine from college days was visiting me last Christmas. He makes an enviable salary by writing code and by maintaining existing code. I recall the following conversation with him.

[Friend] *What kind of research do you do these days?*
[AL] *I am initiating some work on program understanding.*
[F] *What is that?*
[AL] *Well.. you recover the design of a software system from its source code so you know how it does whatever it does.*
[F] *You mean the design of a whole system?*
[AL] *Yes.*
[AL] *Interesting.. why would you want to do that?*
[M] *So that you can understand the system when you are modifying it.*
[AL] *Hmmm.. I can't see how it will help me in my work.*
[M] *Well.. then what would you like?*
[F] *I'd like something that helps me get the job done ... fast.*
[AL] *Which is?*
[F] *Help me in finding what needs to be changed, make the change, and get the hell out.*
[AL] *Well wouldn't the design of the system help then.*
[F] *It will ... but you don't need the design of the whole system. Only the part that needs to be changed.*

Programmers such as my friend, I thought, were the cause of the whole software crisis. It was indeed a shocking moment. I had never before come face to face with such a mindless, irresponsible programmer who was disgracing the entire programming community. I had only read about the effect of their professionalism in delaying projects to bringing the world to the brink of a nuclear disaster.

But then there is a little part in me that has been influenced by Gause & Weinberg's *Exploring requirements* [1]. One reason why, they say, many software projects fail is that the software developed does not solve the user's problem. This happens sometimes because the developers (or analysts) were not *listening* to the user but were instead building what they thought the user needed. As a researcher I

want to develop a tool that would help software maintainers like my disgraceful friend. Whether I like it or not he is my client or user. Since I am still in the requirements analysis stage of my work, I thought, I should at least listen to what he has to say.

I kept pondering over my friend's last statement. My initial reaction to it had been, "How do you find what needs to be changed if you do not understand the whole system". Every time around my friend's statement made more sense. Now I am at a stage where I [almost] do not think of him as a despicable creature. The difference I have is in the strength of the statement. My assertion is: you don't "always" need the design of the whole system to change it "correctly".

How did I come to this conclusion? Over the 6 months following the conversation with my friend I was involved in modifying two software systems: the GNU C Compiler (*gcc*) from Free Software Foundation and the Wisconsin Program Integration System (wpis) from University of Wisconsin*. Both the systems are written in C. Some students and I performed 4 modification exercises: 3 on *gcc* and 1 on *wpis*. Of these 2 were successful, 1 partially successful, and the fourth was aborted. The two successful attempts, 1 each on *gcc* and *wpis*, required the knowledge of a small part of each system. I still do not know how the rest of each of these systems work except the parts modified [and their interfaces]. The exercise that failed required knowledge of a larger part of the system, beyond the scope of my capabilities [and patience]. The exercise that was partially successful resulted into a compromise; we changed the requirements.

The key point is: changes were performed without knowing the whole design. The other two exercises needed knowledge of a larger part of the system but still not of the whole system. In all the cases the design, whatever you may mean by it, of the whole system being worked on was not necessary. And if you are willing to accept the call graph of a program as a form of design, you may like the following. To perform the fourth change we used Brown University's FIELD [2] to see the call graph of a component of *gcc*. Although FIELD provides great features to abstract sets of functions into files and sets of files into directories, it couldn't help us cope with the complexity of *gcc*. This system has about 290,000 lines of code in about 175 files, all in one directory. The sight of the call graph actually frightened my students so much that they requested a different project. They were much more enthusiastic before they saw the call graph.

How do we understand code?

The program modification exercises were not performed to learn about the process of understanding programs. However, at the time the modifications were being performed the researcher in me was observing what the programmer was doing. One observation I made was that I was able to "scan" through the code on-line and zero in on the parts that needed to be modified. In contrast, my students would spend countless number of hours going through the hardcopy listing and reach nowhere.

Permit me to proclaim myself as an *expert*. The other category *novice* commonly used in the literature does not clearly define the qualifications of the students. They are all computer science graduate students who are knowledgeable about 'swap' and 'sort', examples commonly used in empirical studies of programmers. Let me call the students as *interns*. There is clearly a difference between how an *expert* and an *intern* understands "large" software system. It is beyond what can be captured by the *chunking*

* The experience cited here relates to version 1.40 of *gcc* and the first release of *wpis*.

```
static struct compiler default_compilers[] =
{
  {".c", "@c"},
  {"@c",
   "cpp -lang-c %{nostdinc} %{C} %{v} %{A*} %{D*} %{U*} %{I*} %{i*} %{P}\
  %{C:%{!E:%eGNU C does not support -C without using -E}}\
  %{M} %{MM} %{MD:-MD %b.d} %{MMD:-MMD %b.d}\
```

Figure 1 Extract from GNU's C preprocessor. The data structure contains a "program" that describes what subprocesses would be invoked by the compiler, their order, input, and outputs. The language used is internal to GNU C Compiler. (This approach of encoding program control information in a data structure is sometimes also called as table-driven programming).
© 1992 Free Software Foundation, Inc.

model of program understanding which says that experts look at patterns in code and match them with mental models of computer science or domain concepts [3, 4, 5, 6].

When I was scanning the code I did not read the code but was actually "looking for" specific clues. For instance, searching for the place where the *gcc* preprocessor expands macros and the place where it processes "#define" statements. Since a particular modification was related to these two features I, at least to start with, was not interested in how other features were implemented. The function names *macroexpand()* and *do_define()* caught my attention and were sufficient clues. To summarize, my search was influenced by the knowledge of what the system does and by a hypothesis of how it did it.

There were some instances when my search failed because the specific behavior was not implemented in the ways I had hypothesized. For instance, when searching for the place where *gcc* invokes the preprocessor I was looking for a call to *execv()* or *fork()* with a parameter that had the string "cpp" or "CPP" in it. I did not find what I was looking for but instead found a piece of text, such as that in Figure 1. A scan, and not a complete understanding of this text, indicated that *gcc* has its own language, called "spec", to encode the flow of control between the subprocesses it invokes. The information about invoking the C preprocessor is encoded in this language and maintained in a table. This evidence required me to throw away my old hypothesis and create a new one.

I read a piece of code in detail only after I had located the "places" where a specific behavior was implemented. Subsequently I visited only those segments of code which either called these functions or were called by it. Further, even when understanding code at an individual function level, most often, I was "looking for" something (a top-down approach) as against "reading and understanding" (a bottom up approach).

Till then my knowledge of work in program understanding was limited to the chunking model [7] which implies a bottom up approach. Excited about my analysis and like a true researcher I headed to the library. Look what I found. Ruven Brooks in his paper [8] gives a theory of program comprehension whose major points are summarized as follows:

1. The programming process is one of constructing mappings from a problem domain, possibly through several intermediate domains, into the programming domain.
2. Comprehending a program involves reconstructing part or all of these mappings.
3. This reconstruction process is expectation driven by the creation, confirmation, and refinement of hypotheses.

His theory, to the best of my knowledge, has not been validated experimentally but he used a "Theory Demonstration" approach to support it.

Finding a work that almost a decade ago did what is still an idea in your mind is both discouraging and encouraging. It is discouraging because you feel someone beat you to it. It is encouraging because it reassures you that your mind generates ideas that are credible. In any case, I was pleased because I did not think I could formulate the theory in as meticulous a way as Brooks did. Besides, I don't think all is lost. Given the difficulty in designing, performing, and analyzing statistically valid experiments for cognitive models, my experience may be taken as a validation of Brooks' theory. Since I was not familiar with his theory before I thought of the model, in my opinion, is a strong argument in its favor.

Let us take a closer look at the theory. So far my experiences only support the third argument which implies that the process of program understanding is top-down, i.e. one formulates a hypothesis as to how a program does what it does and then looks at the program to confirm the hypothesis. If the hypothesis fails then the programmer generates a new hypothesis. And as Brooks argues, the hypothesis generation process is limited by the experience of the individuals. One resorts to the bottom up approach, i.e. read and understand the code, only in the extreme case when the programmer is utterly unfamiliar with the domain.

The first statement implies: "When a programmer completely understands a program, what he knows can be described as a succession of knowledge domains that bridge between the problem being solved and the program in execution" [9]. The succession of domains may vary from problem to problem. An example would be a top level domain that expresses the functionality of the program; a second domain that associates constraints to these functionalities (example size of some data); a third domain that associates an algorithm to achieve a function, a fourth domain that describes a real world object as mathematical objects (such as trees, sets, matrices); a fifth domain that maps the algorithms and mathematical objects to source code.

Notice that this statement implies how knowledge is organized in a the programmer's mind. I would not know how to verify this experimentally. But I think that from this statement one can infer that if a program's source code (and its documentation) is designed and organized to reflect the succession of such domains then such a program would be easy to understand. Again, my experience with modifying *wpis* when compared with that of modifying *gcc* validates that. The *wpis* modification exercise was smooth and effortless but that of modifying *gcc* was telling. On comparing the two system's one may see that *wpis'* documentation and organization of source code reflects the succession of domains that Brooks talks about. The following discussion performs a detailed comparison.

I have already mentioned about the physical organization of *gcc* source on the machine. The top level directory of *gcc* contains 334 files. This includes 115 .c files, 68 .h files, and 4 .y files; README, *texinfo*, and man page documentation files; configuration files to install on a dozen or more architectures on which *gcc* can be installed and some Makefiles. The Makefiles contain targets for several executables and libraries that are part of the *gcc* system. The source code totals about 290,000 lines of text. When the complete system is generated approximately 115 .o files are created in the top level directory itself.

Contrast this with *wpis* whose top level contains 7 directories and a Makefile. There is a directory *src* that contains the source. The source code totals 41,000 lines of text but it is not all contained in one directory. The *src* directory is further divided into 8 subdirectories which in turn have another 1 or 2 levels of directories before the actual source files are found. The lower most directories house code for a specific module. Some of the modules implement general purpose data structures such as

```
static void
macroexpand (hp, op)
     HASHNODE *hp;
     FILE_BUF *op;
{
     ..... 92 lines deleted

  /* Compute length in characters of the macro's expansion.
      Also count number of times each arg is used.  */
 xbuf_len = defn->length;
 for (ap = defn->pattern; ap != NULL; ap = ap->next) {
 if (ap->stringify)
   xbuf_len += args[ap->argno].stringified_length;
 else if (ap->raw_before || ap->raw_after || traditional)
   xbuf_len += args[ap->argno].raw_length;
 else
   xbuf_len += args[ap->argno].expand_length;

 if (args[ap->argno].use_count < 10)
   args[ap->argno].use_count++;
  }

      ...... 181 lines deleted
}
```

Figure 2 Parts of a function extracted from GNU C Compiler.
© 1992 Free Software foundation, Inc.

graph, set, queue, and *sequence*. The directories for these modules are combined in a directory called *packages*. Similarly, there are modules that implement complex data structures, such as *pdg_module* for implementing program *dependence graph*, and complex algorithms, such as *program slicing*, their directories are combined into a directory called *algorithm*. There are separate modules for interfacing with the windowing system and other software systems needed to use *wpis*.

The Makefile of *wpis* is written such that on compilation the .o files are generated in directories different from the source files. The document files are in separate directories and so are examples to demonstrate the system with.

In my opinion, the organization of *wpis* source code reflects the domains that in Brooks' terms a programmer may reconstruct. The subdirectories *graph, set, queue*, and *sequence* correspond to the mathematical objects that are placed in the directory *packages.* The directory *pdg_module* is also an object in the realm of the implementation and is defined using the basic mathematical objects. It is kept separately along with other modules defining concepts at the same level.

To understand *gcc*'s implementation you have to start with identifying the set of files, if any, that contains code to perform the lexical analysis, parsing, code generation, optimization tasks; i.e. reconstruct the mapping between conceptual domains and files. Since there are literally hundreds of files in the source directory (and even more if one has the .o files around) the reconstruction of domains and their mappings is not a simple task.

```
static PROCEDURE VisitVertexForForwardSlice( vid, g )
VERTEX_ID vid;
PDG g;
{
  PDG_VERTEX v;
  PROCEDURE ProcessSuccessorForForwardSlice();

  v = pdg_vertex_retrieve(g, vid);
  if (v == PDG_VERTEX_NULL) return(SUCCESS);
  if (pdg_vertex_mark(v)) return(SUCCESS);
  pdg_vertex_mark_set(v);
  set_for_each1(pdg_vertex_loop_independent_targets(v),
    ProcessSuccessorForForwardSlice, g);
  set_for_each1(pdg_vertex_loop_carried_targets(v),
    ProcessSuccessorForForwardSlice, g);
  set_for_each1(pdg_vertex_control_targets_true(v),
    VisitVertexForForwardSlice, g);
  set_for_each1(pdg_vertex_control_targets_false(v),
    VisitVertexForForwardSlice, g);
  return(SUCCESS);
}
```

Figure 3 A function extracted from Wisconsin Program Integration System
© 1989 Thomas W. Reps and University of Wisconsin, Madison.

Similar differences exist in the source code of the two systems too. For instance, look at the code segments in Figures 2 and 3. These are two functions from each system that were located as candidates for introducing the changes I wanted. In each case the name of the procedure reflects the externally observed behavior it is related to. Once you know the external behavior you wish to change, it is easy to locate these functions. Figure 2 only shows a fraction of the 294 lines function. This function operates on the specific data structures *directly*. The comments relate the code segments following them to the externally perceived behavior. Figure 3 on the other hand shows the whole function. It has no comment within the body of the code and it also does not operate on the data object directly. Instead it uses operators whose names define the mathematical operations. Each statement corresponds to a unit subtask at a level higher than the specific implementation of the data structures.

In Figure 2 the comments are supposed to help you in creating a mapping from a higher level domain to the code. In contrast in Figure 3 the code itself reflects this mapping. So far, besides myself, I have had three students work on *gcc* and one on *wpis.* The one who has worked with *wpis* thinks that modifying other people's code is a piece of cake. The other three were praying "Beam me up Scotty" before I changed their project.

There is another factor that has influenced the modification exercises. In case of *wpis* the various algorithms that it implements are available in published literature. The student and I had thoroughly understood the algorithms before attempting to understand the code. In a sense we had a set of hypotheses before we looked at the code. There were two instances where our hypotheses were violated. In one case we were looking at some code that in our opinion could not even be compiled. After spending a lot of time we contacted its author to explain to us what that piece of code did. We were told that that piece was "dead code". In the second instance we had a difficult time understanding an algorithm that required

traversal of a special type of graph. The traversal mechanism in the implementation was significantly different from that in the published work. We spent a few hours trying to map the published algorithm to the code and when we couldn't, we assumed that our understanding of the program was at fault. We then resorted to the bottom-up approach and realized the discrepancy.

In case of *gcc* our hypothesis formation was also limited by our knowledge of compiler construction. For instance, most textbooks on this subject suggest splitting a compiling task into a sequence of activities: lexical analysis, parsing, abstract machine code generation, optimization, and actual machine code generation. But that division we learnt is for teaching the concepts of compiling. The developers of *gcc*, whom I classify as *gurus*, have interleaved several of these tasks. And we learned it the hard way because we ignored an inconspicuous statement in its documentation stating that fact. There is a good chance that the algorithms used by *gcc* are published. But because of our ignorance we were expecting a task decomposition suggested in textbooks. The knowledge of interleaved implementation of tasks helped us in creating the correct domain reflecting task decomposition of *gcc*. But in the two weeks that the students spent on understanding that part they were still unable to map this decomposition to the actual code and hence we gave up one of the exercises.

Now let us look at the second statement made by Brooks: comprehending a program involves reconstructing part or all of these mappings. The phrase "part or all" provides the possibility that one may only understand part of the program. Which is similar to my observation: you do not always need to understand the design of the whole system.

The question that now remains is: how widely accepted is Brooks' theory? It turns out the Brooks is not alone. Soloway et. al. have identified a strategy used by professional programmers in understanding code. They call it *inquiry episode* which is a cycle of: "read" code; ask "question" about it; "conjecture" an answer; and "search" documentation for confirmation [6]. Using their terms then Brooks' model proposes a cycle of: conjecture, search, read, and question. And since they are cycles, they are identical.

Summary

Let me now summarize how 'I' understand code, the factors that influence 'my' ability to understand it, and some challenges in building tools to aid the understanding process. Since I have only analyzed my mental processes during understanding programs I can only talk about myself.

First of all I do make hypotheses of how a system is implemented even before looking at it. The hypotheses are influenced by my knowledge of the domain of application and/or my ability to draw an analogy between the problem it solves and one that I may have experience with. If I do not have adequate domain experience, I find it useful to first read material about the domain. The material may be books, articles, user manuals, or other documents. Source code is a very poor substitute for these. It helps if the system documentation contain references to such material. When I do not know about the domain and also do not have any supporting documentation I resort to the bottom up approach to reading the code to understand it. My reading of the code is neither linear nor complete, i.e. very rarely do I understand all parts of a function. Instead I scan through the code identifying patterns and understanding them; putting those patterns together to get the bigger picture.

There are times when the changes I introduce do not affect a software system's behavior as I expect them to. In most such cases I have found that the changes I made were based on an incorrect hypothesis. The fault being that I assumed the hypothesis to be right without verifying it. Sometimes this happens

when I lose my objectivity and do not reject my working hypothesis even after coming across information in the source code or documentation that violates it.

My ability to understand a system's implementation is influenced by the availability of additional documentation, the organization of its source code on the machine, as well its actual design. It is easy to understand an implementation that uses the same terminology, symbols, and algorithms stated in its documentation. Most often the conflict arises when this is not the case. For instance, when an equivalent but different algorithm is used. It is harder to understand a system that uses non-traditional (read non-textbook) designs. It is also hard when the system's implementation does not reflect the succession of domain knowledge useful in understanding it. Providing comments to aid in recreation of levels is useful. I however prefer that the levels be reflected in a combination of physical organization of the source code as well as in its modular decomposition.

Maintenance requests are usually phrased in terms of a program's requirements not its internal working. One of the first task a maintenance programmer does is to locate pieces of code that implement a functionality. Biggerstaff et. al. term this as the concept assignment problem [10]. Although this has been recognized by practitioners for quite sometime [11], very little research has been done to support this activity. Wilde and Gust's recent work is a step in that direction [12]. They propose tracing a program on sample test data that enumerate its features and using the trace to perform a mapping between its components and features. Interestingly, this is the only work I know of that proposes using information collected from executing a program for understading it.

At the moment I rely on *grep, more*, and *emacs* to locate pieces of code that implement a functionality. Since I do not deal with programs that have millions of lines of code, these *ad hoc* methods suffice. In my experience tools that display call graphs have *not* been very useful. Such cross-reference information may instead be provided through database queries or hypertext oriented browsers. What I would really like is a tool that recovers the design that is most natural to express the type of problem I am working with. For instance, for a process intensive application I would like the state-transition diagram. If the application is data intensive I would like the entity-relationship diagrams of its data model. If the application implements its own language (as in Figure 1) and then interprets programs written in it; I'd like the tool to help me in understanding those programs.

The program understanding tool should integrate with the system building process. If there is a *Makefile* that is used to build a system, the tool should extract information from it rather than requiring me to provide it. This will especially be appreciated if I am working on someone else's code. In this situation I have no clue as to what files and compilation options are needed to generate the system. I recently bought a rather expensive reverse engineering tool that does not integrate with *Makefile*s. Its manuals are now collecting dust because the work required to start using it is enormous. In contrast, Brown University's FIELD [2], besides being free, only requires the name of the executable file to figure out [almost] all that is needed to extract information about the system. We have put this system to extensive use.

Finally, I do not enjoy understanding someone else's code unless I want to do something with the knowledge I so acquire. Most often I will use the knowledge to add, remove, or change its behavior. In this case it will be nice if the program understanding tool does not require me to recover the design of the whole system.

Bibliography

[1] D. C. Gause and G. M. Weinberg. *Exploring Requirements: Quality Before Design*. Dorset House Publishing, New York, 1989.

[2] S. P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.

[3] V. R. Basili and H. D. Mills. Understanding and documenting programs. *IEEE Trans. Softw. Eng.*, SE-8(3):270–283, 1982.

[4] C. Rich. *Inspection methods in programming*. PhD thesis, M. I. T. Artificial Intelligence Laboratory, Cambridge, MA, 1981.

[5] B. Shneiderman and R. Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information Services*, 7:219–239, 1979.

[6] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert. Designing documentation to compensate for delocalized plans. *Commun. ACM*, 13(11):1259–1267, Nov. 1988.

[7] B. Shneiderman. *Software Psychology*. Winthrop Publishers, Cambridge, MA, 1980.

[8] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[9] R. Brooks. Using a behavioral theory of program comprehension in software engineering. In *3rd International Conference on Software Engineering*, pages 196–201, Los Alamitos, 1978. IEEE Computer Society Press.

[10] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of Conference on Reverse Engineering*, page to appear, May 1993.

[11] N. Zvegintzov. Process of software change (keynote speech). 3rd Reverse Engineering Forum, Burlington, MA, Sept. 1992.

[12] N. Wilde and T. Gust. Locating user functionality in old code. In *Proceedings of Conference on Software Maintenance, 1992*, pages 200–205, Los Alamitos, 1992. IEEE Computer Society Press.