Analysis of experiences with modifying computer programs

Arun Lakhotia

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

Available as: CACS TR-93-5-6

REVISION HISTORY: 12/1/93, 3/22/1993

Abstract

The paper analyzes the author's experience with modifying large, real-world programs written by other programmers. It finds that Brooks' domain and programming knowledge based hypothesis-test-refine paradigm explains the author's approach to understanding programs and the differences in performance in comparison with his students. Zvegintov's 9–step process of change is found to be a good first level decomposition of the (physical) tasks performed when making corrective changes to a software system.

The paper also makes some new observations. Besides modularity and levels of abstractions, the organization of source code in hierarchy of directories also has influence on the ease of locating code segments relevant to a change request. The functionality of a program is not only understood from its documentation but also by executing it and inferring relations between its inputs and outputs; an approach analogous to concept identification. When introducing a new function in an existing program, a programmer attempts to find subproblems that have been solved by other parts of the program so as to mimic their solutions. Quite often this means copying large code segments. However, when deleting a function, the code implementing it is not destroyed, only execution paths leading to it are disconnected; leaving behind dead-code. The replicated and dead code segments are major contributors to the difficulty in understanding and modifying programs.

1 Introduction

How does an expert or a novice programmer understand and modify computer programs, especially those not developed by him? What are the cognitive processes active in a programmer's mind when performing these activities? What are the tasks performed during program modification? Answers to these questions are central to our concern over the enormous cost of maintaining software systems. They could explain the differences between an expert and a novice programmer's capabilities in maintaining software systems. This would lead to the development of software tools, documentation standards, and training programs to augment the skills of the novice thereby increasing their productivity.

Software maintenance heavily relies on the programmers' ability to comprehend programs [7]. This observation has prompted several researchers to investigate the processes involved in understanding programs. Of interest to this paper are works from two disparate communities: academic researchers in computer science and psychology, such as [1, 3, 18, 19], and practitioners and managers in the industry, such as [22, 23, 25].

The papers [3, 18] propose theoretical models of cognitive processes involved in program understanding. These models have been subjected to experimental analysis by other researchers [9, 20]. The experiments are typically performed in controlled environments using programs under 500 lines of code. Whether the conclusions drawn from such experiments can explain the processes involved in understanding larger programs is debatable [2]. In contrast, [22, 23, 25] propose steps taken in making modifications to real-world programs. The steps have been assimilated from years of experience with participating in the maintenance of real-world software systems. The steps provide rules-of-thumb guideline from one programmer (or manager) to another on how to decompose and order the tasks during program modification and what mistakes to avoid. Their efficacy has not been validated.

This paper attempts to bridge the works of these two disparate communities. It is centered around the author's analysis of program modification exercises that he and his students have performed. The exercises were not performed to learn about the mental processes or specific tasks involved in modifying programs; they were performed for the intent of introducing new behavior in two real-world programs. The steps in the exercise were not recorded. At the time when the modification exercises were performed the author was looking for open problems in developing tools for supporting software maintenance. He was consciously observing his own actions and that of his students so as to identify needs that may be filled by a software tool.

Since performing controlled experiments on real-world programs is extremely difficult, most experimental and observational studies on programmer behavior have used "toy" programs. Whether their results are generalizable to real-world programming problems is questionable [2]. There in lies the importance of this paper. It has evolved from the observation of real-world program modification exercises performed by the author. Like other observational techniques, such as protocol analysis, introspection may be used to develop initial hypotheses about human behavior [8, 6]; these hypotheses may later be validated empirically. This paper makes several new observations that could be the subject of experimental validation. The observations of this paper should not be taken to construe theory of programmer behavior.

The rest of the paper is organized as follows. The next section summarizes the works relevant to this research. Section 3 gives a narrative of four program modification exercises performed on two real world programs. Our observations from each exercise are given in separate subsections. To understand the programming problem and solution one may need specialized knowledge about the software and the

problem. Reader not conversant with or interested in these details may directly go to the subsections describing our observations. Section 4 gives a comparative analysis of the factors influencing our ability to modify the two systems. It also analyses the potential causes of difference between the author's and the students' capabilities to understand and modify others' code. Section 5 summarizes our observations.

2 Related works

This section gives an overview of the research efforts in studying the physical processes [1, 22, 23, 25] and the cognitive processes [3, 18, 21] involved in program comprehension and modifications. We consider physical processes as externally observable steps carried out when modifying a software. In contrast, cognitive processes are processes active within the mind.

2.1 Physical process models Wachtel [22, 23] and Zvegintov [25] give steps involved in modifying programs in an industrial environment. Their steps are rules-of-thumb guidelines accumulated from years of experience maintaining software systems. Wachtel observes that software modification is initiated by a "change request" and terminates with the "release" of the software. Between these two states is where the actual modification takes place. Zvegintov proposes a nine step process that links these two states:

1. Get an *overview of the functionality of the current system*. What does the system do? What is it used for? etc.
2. Get an *overview of its implementation*. Understand the high level components that come into play during execution. For instance, the machines, equipments, people, and software items that constitute this system.
3. *Outline the request*. Identify the functionality that would be altered as a result of this change. Define the inputs and outputs for a set of cases that would be changed, a set of related cases that would not change, and a set of boundary cases.
4. *Trace the request*. Identify the parts of the software system that need to be modified for implementing the change. This is the first step where the source code is inspected.
5. *Cover the request*. Make sure that all the parts that need to be modified have been identified (in the event that code has been duplicated to handle similar conditions in different situations).
6. *Design the change*.
7. *Implement the change*.
8. *Ripple the change*. Code "downstream" to the modified code may be exposed to data that it did not before modification. It may therefore create conditions that did not exist in the earlier version. Follow the execution to identify these locations and modify them as well.
9. *Prove the change*.

Zvegintov's 9-steps define a top-down process. Steps 1 to 4 may all be considered as understanding the system. This includes understanding issues outside the source code and documentation. The inspection of the source code is deferred till the fourth step. Zvegintov's process of change is centered around the change request. This request influences which part of the program is understood. As a corollary, if there is no change request, there is no need to understand a program.

Basili and Mills have proposed a bottom-up approach [1] for understanding programs. Their approach involves transforming the program through a succession of abstract representations and finally terminating

into a PDL, an abstract language, specification of the program. Their central concern is understanding the program independent of the change request.

2.2 Cognitive processes in program modification Shneiderman [18] has proposed that understanding a program implies creation of "multileveled internal semantic structure to represent the program". The highest level in this structure encodes what the program does while the lower level may recognize familiar sequences of algorithms or statements. This multilevel structure is created bottom-up. A programmer recognizes the function of groups of statements, pieces them together to form ever large chunks until the entire program is comprehended.

According to Shneiderman, the first step when modifying a program is the development of the internal semantics representing program. The change request is applied to the internal semantic representation and then propagated to the actual program. Shneiderman is not explicit on whether partial understanding of system is permitted or whether the change request has any influence on how the system is understood.

Brooks [3] has proposed a competing theory of the cognitive processes involved in program comprehension. He says that during programming one creates mappings from a problem domain to the programming domain, passing through several intermediate domains. Program comprehension is the reconstruction of this mapping performed by a hypothesis-test process. A programmer first creates a hypothesis about the program's behavior and then looks for evidence to support or reject the hypothesis. If the hypothesis fails a new hypothesis is generated, and the process continues. In the extreme situation when the programmer is unable to create any more hypotheses, he resorts to the bottom up approach, i.e. read and understand the code.

Brooks permits incomplete understanding of the program in that all the mappings between levels of domains may not be constructed in order to carry out a task. He contends that comprehension may be affected by the modification being performed. To test a hypothesis, the programmer may search for "beacons" to locate relevant code. Brooks has only theorized about the comprehension process when modifying programs. He has not suggested the process by which modifications are carried out. He emphasizes on the importance of domain knowledge in the understanding process. That is, if a programmer is not knowledgeable about the program's domain, it will make it harder for him to understand the code.

Soloway et. al. [21], studied the protocols of programmers modifying a piece of code. They found that programmer's went through cycles of *reading* the code, raising *questions* about it, making a *conjecture* about it, and *searching* code for answers. They termed the read-question-conjecture-search cycle as an "inquiry episode". Since it is a cycle, it could as well be written as question-conjecture-search-read, which then maps well with Brooks' model.

3 Narrative of modification exercises

This section gives a narrative of four program modification exercises, referred to as MOD I - IV, that the author and his students performed. It reconstructs the steps taken and some of the protocols exchanged during these exercises. This reconstruction of protocol is based on recollection of the modification exercises. They preserve the essence of the communications between the author and the students, though they may not be verbatim. It must also be emphasized that the exercises were **not** performed to study human behavior, but were done as part of a separate research effort.

3.1 Subjects The subject of the modification exercises were four second year graduate students, referred to as B, C, D, E, and the author, referred to as Subject A. Subject A has had considerable experience with modifying programs written by others. Subjects B to E were experienced with developing programs. They did not have experience with maintaining programs.

3.2 Material for modification exercise The software systems modified were: 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. Of the four modifications, three were performed on GCC and one on WPIS.

3.2.1 Material 1: GCC This system has about 290,000 lines of code in about 175 files, all maintained in one directory. 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 it on a dozen or more architectures and some Makefiles containing procedures for generating and installing the system. The Makefiles contained procedures to generate several executables and libraries that are part of the GCC system. The source code (i.e. the .c, .h, and .y files but not the documentation and other files) 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.

3.2.2 Material 2: WPIS. At the top level this system contains seven 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 eight subdirectories which in turn have another one or two 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 *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.

In WPIS each module has a separate Makefile containing instructions to compile that module. These are all tied together by a hierarchy of Makefiles communicating with each other by a large set of "macros". The Makefiles of WPIS are written such that on compilation the .o files are generated in directories different from the source files. The documentation files are in separate directories and so are examples used to demonstrate the system

3.3 Purpose of the modification exercises The modification of GCC was initiated in order to experiment with architecture recovery techniques [5, 11, 16, 15]. The three exercises, described later, collectively would have modified the GCC to output some cross reference information needed by such a browsing tool. While the first two exercise were completed, the third was not because by then we found a public domain tool, FIELD from Brown University [17], that performed most of what we wanted. The last modification exercise which, as stated later, was more complex than the first two. Since it was not cost-effective to complete that exercise, it was abandoned.

* The experience cited here relates to version 1.40 of *GCC* and the first release of *WPIS.*

The modification of WPIS was performed to experiment with an algorithm for interprocedural program slicing [14]. This algorithm was a variation of an algorithm due to [10]. This exercise was successful. The reason we chose WPIS to develop the slicing algorithm was because it already implemented the construction of "program dependence graph", an internal representation of programs. This algorithm for constructing this representation is very intricate and we did not feel it worthwhile to implement it when WPIS was available.

More recently we have purchased the Software Refinery toolset from Kestrel Institute and Refine/C extensions from Reasoning Systems, Inc. These tools provide a much better platform for our experimental research. They have also completely removed our dependence on modifying other software systems. As a result most of our prototype development activity has been moved to this platform.

3.4 Procedures MOD I and II were performed in Fall 91 by Subjects A and B. Subject B did the modifications as a Research Assistant and hence was paid for the work. The two subjects worked in close cooperation.

MOD III was performed in the first two weeks of Summer 92 by Subjects C and D as a part of a course project. Subject A played an advisory role only.

MOD IV was performed by Subject E in Fall 92 as a part of a course project. Subject A supervised all steps of the exercise.

3.5 Exercise MOD I This exercise enumerates several of the assertions we make about programmer behavior. Reader uninterested in details of the experiment may prefer to skip to Section 3.5.3.

3.5.1 Objective of MOD I Modify GCC's "preprocessor" such that when it processes a "`#define`" directive it outputs, along with its normal output, the `MACRO` symbol, the line at which the definition starts, and the line at which it ends. The format of the output should be designed so as not to "break" subsequent parsing and code generation activities.

3.5.2 Reconstruction of steps for MOD I The tasks performed to carry out the modification can be analyzed in terms of Zvegintov's 9-steps process of change [25]. The reconstruction is created by mapping the tasks to Zvegintov's steps. The presentation has been simplified to show this mapping. In the actual exercise, the steps were not always carried out in the order presented.

OVERVIEW OF SYSTEM FUNCTIONALITY. Subject B was given a short on-line lesson on GCC. It has three executable commands *gcc*, *cpp*, and *cc1*; the *gcc* is the front end used for compiling files; it invokes *cpp* and *cc1*; the *cpp* preprocesses the "#" directives (including "#define"); the *cc1* takes its input from *cpp* and compiles the program. This decomposition is common for C compilers. The lesson was based on the Subject A's prior knowledge of GCC. Subject B had neither used GCC before nor was aware of decomposition of C compilers.

OVERVIEW OF GCC'S IMPLEMENTATION. We browsed through GNU sources to get a feel of it. We learnt its installation procedures and experimented with installing it. The prime question in our mind was:

[M1-1] Which of the (115 .c, 68 .h, and 4 .y) files were used for creating *cpp*?

The answer was found from the procedure to generate *cpp* in the Makefile. The next question now was:

[M1-2] What is the format of its output?

This was important to design an output message that contained the necessary information, was consistent with the outputting conventions of *cpp*, and would not break *cc1*. The Manual pages for *cpp* contain some information about the format. This was not sufficient for us. So we executed *cpp* and attempted to infer what it did. This required creating some sample data, executing *cpp* with it, and analyzing its output. The analysis went about by reasoning *forward* - what the output would be used for in the context of what *GCC* (as a whole system) did and *backward* – what information did *cc1*, the parser, need from *cpp* so it could carry out its function.*

OUTLINE THE REQUEST. We now defined the syntax of the output and verified that it would not interfere with *cc1*, the parser. The verification required creating "mock" inputs for *cc1* and executing *cc1* directly instead of through *gcc*.

TRACE THE REQUEST. The question at hand was:

[M1-3] Where is `#define` processed in the *cpp* source code?

We did not attempt to trace the request by symbolic execution of the program. Instead we scanned the program on-line looking for some clues. The following piece of text caught our attention:

```
int do_define (), do_line (), do_include (), do_undef (), do_error (),
  do_pragma (), do_if (), do_xifdef (), do_else (),
  do_elif (), do_endif (), do_sccs (), do_once ();
```

The *cpp* processes commands such as `#define`, `#line`, `#include`, `#undef`. It was therefore *assumed* that the list, above, gave the name of functions where these commands were processed. We looked at the code for `do_define()`, extracted below:

```
/* Process a #define command.
... comments about arguments deleted ... */
do_define (buf, limit, op, keyword)
     U_CHAR *buf, *limit;
     FILE_BUF *op;
     struct directive *keyword;
{
  U_CHAR *bp;   /* temp ptr into input buffer */
  U_CHAR *symname;   /* remember where symbol name starts */
  int sym_length;    /* and how long it is */
 .... lots of code deleted ...
```

The first line validated our assumption that `do_define()` processes `#define`. The last two comments provided information to our next question. They told us that the name of the macro will be available in `symname` and `sym_length`. Also, while we were scanning the text to find the code for `do_define()`, we noticed the string:

`ip->lineno`.

This string was interesting because we have to output line numbers and the field `lineno` was therefore relevant. We assumed that `lineno` field was used to keep the number of the line of the input being processed and that `ip` referred to the input file. These were later confirmed from the comments in the code. Following similar questions and clues we found that the two line numbers we needed were available in the function `handle_directives()`, the caller of `do_define`().

* Note that the forward and backward reasoning we performed is different from forward and backward program slicing [24]. In program slicing the analysis is performed over a program's source code. Our analysis was performed over a program's input and output behavior. Besides, *cpp* and *cc1* are two different executable programs. Analysis between interactions of such programs is not covered by program slicing.

We now had located the places in the code where the name of the macro and the two line numbers as well as the data structures containing them were available. These are the places that would have to be changed.

COVER THE REQUEST. The comments in `handle_directives()` said it processed "#" directives. We could not think of a need for more than one function to process directives and hence did not worry about checking it. We did check if `do_define()` was called from any other function. It was called from `make_definition()` to create some initial definitions. This was a surprise to us even though it corresponds to a feature documented in the GCC Manual pages. Evidently, since the feature never interested us we must have ignored it.

DESIGN AND IMPLEMENT THE CHANGE. Even though we had the three pieces of information needed to be output, there was still one more problem. The two line numbers and the macro name were not available in the same function; the line numbers were available in `handle_directives()` but the macro name was available in `do_define()`. To implement the change, one could either split the task of outputting the needed information across these two procedures or propagate the information to a location in either of the procedures. We generated various alternatives and evaluated them:

[M1-4] If we output in `do_define()` it will also output the information when `do_define()` is called from `make_definition()`.

[M1-5] If `do_define()` returns the macro name it has to allocate space for the symbol or the caller should pass an argument. But these options will require changing the interface of `do_define()`

[M1-6] The `handle_directives()` function could duplicate from `do_define()` the search for finding the macro name. This will not require any changes in the interfaces and hence has low chances of interfering with other behaviors.

We chose M1-6. Other options would have required changing the interfaces and we did not know what the repercussions would be. We still had one more question left: What conventions do we need to follow to output the informations? We knew that *cpp* outputs special information when processing `#include` directive. Hence the solution was:

[M1-7] Let's see what *cpp* does for processing "`#include`". We could "mimic" the output technique it uses there.

We copied the code for outputting information and customized it to our needs.

RIPPLE AND PROVE THE CHANGE. We did not do any special test for "rippling" the change. To prove the change we compiled some of our C programs to see if the compiler behaved as we desired.

3.5.3 Analysis of MOD I We make the following observations from the reconstruction above.

The various tasks performed while modifying program can be mapped to the 9 steps proposed by Zvegintov [25]. This correspondence of steps for the above exercise is provided with the reconstruction.

The functionality of a system is understood not only from its documentation, but also from executing it. To find the format of *cpp*'s output (M1-2), we executed the program with various inputs and related them with the outputs. To interpret the information contained in the output, we needed the knowledge of what information *cpp*, the preprocessor, needed to provide to *cc1*, the parser. This interpretation was supported by arguments based on programming knowledge and on knowledge about compilers.

Our method of understanding code may be explained using Brooks' theory [3]. The sources used to generate *cpp* add up to about 10,000 lines of C code. The whole exercise was performed on-line,

without any hardcopy. Hence a complete, bottom-up understanding, even subconsciously, is ruled out. Instead as the reconstruction following M1-3 shows, the symbols and the comments in the program were used to locate the code segments that were relevant to the change. The symbols acted as "beacons" that helped us in "homing" in on the relevant code. The comments were the first source of evidence supporting the hypothesis.

A program of the size of *cpp* has a multitude of symbols. Which symbols act as beacons and why? We believe that *whether a symbol acts as a beacon depends on the relevance of the information to the activity being performed.* The beacons we used, `do_define()` and `ip->lineno` (M1-3), were meaningful in the context of the change request. An expert programmer is able to scan the text, identify the symbols that are relevant to the change request, and filter out irrelevant symbols.

The extent to which a program is understood depends on the "amount" of functionality of interest to the programmer. We only wanted to alter the processing of one of many commands that *cpp* processes. As a result we focussed on how that command was implemented. Our change was successful because in *cpp* each "#" directive is processed by a separate function. It would have been much harder otherwise.

New features may be added by "mimicking" or duplicating code for subproblems previously solved by the program. M1-6 and M1-7 give two different reasons. In M1-6 the decision to duplicate code for "searching the macro name" is made based on trade-offs between various alternative designs. However, in M1-7 the suggestion to "mimic" the solution to "output information" is based on the knowledge that when processing `#include` directive *cpp* outputs information and that the two subproblems of outputting information are similar. When the code being mimicked is more than just a function call it leads to replicated code segments, a major problem in maintaining large software systems.

3.6 Exercise MOD II

This exercise exhibits the failure of an initial hypothesis about program behavior and subsequent "search" for information to generate new hypotheses. It shows how lack of experience with the design strategy employed by a program may make its comprehension harder. The reconstruction is restricted to highlighting these points.

3.6.1 Objective of MOD II

Modify the GNU front end, *gcc*, to invoke the modified C preprocessor, *cie-cpp*, instead of *cpp*.

3.6.2 Reconstruction of steps for MOD II

After familiarizing with the relevant source code files, Subject B was asked to:

[M2-1] Find the call to `execv()` or `fork()` that invoke "cpp" and modify it to invoke "cie-cpp".

This directive was based on the hypothesis that:

[M2-2] The functions `execv()` or `fork()` are used to execute other programs. The name of the program to be executed is provided as an argument. Since "*cpp*" is a separate program, there must be a call to one of these functions containing "cpp" as a substring.

Subject B returned after a few days saying he could not find any such call. Subject A was surprised and decided to scan the text, searching for "cpp" and "CPP". The search led to the code segment extracted in Figure 1. Subject A immediately responded:

[M2-3] Pretty neat, the information about subprocess invokation is encoded in table.

```
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 Compiler front-end. 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 some times also called as table-driven programming).

Subject B had also scanned this text but could not make any inference. A short lecture on table-driven programming was sufficient to bring him on par.

3.6.3 Analysis of MOD II *A programmer may scan the code in search of information that may be used to create hypotheses about a program's design.* Scanning the code is different from "reading" the code to understand each statement. It is also different from tracing control flow paths. Scanning a code means flipping through text in search of "beacons". Such a scan may also be done by searching for occurrences of certain "words" or character sequences of relevance to the context.

Notice that scanning code to find some "beacon" is bottom-up though searching for some character sequence is top-down.

An unfamiliar or uncommon design strategy is hard to understand. Subject B was not familiar (or did not have experience) with table-driven programming. As a result the code segment in Figure 1 did not interest him. Had he symbolically executed the code, from the start, he may have understood what this table was used for. The feasibility of such in depth reading is however questionable.

3.7 Exercise MOD III

This exercise once again shows the effect of a wrong hypothesis. It also provides a case where scanning the code did not reveal any information and reading or tracing the code to understand it was impossible. The clue about the design strategy used was available in one statement in the document which we happened to have ignored. The reconstruction is limited to highlighting these points.

3.7.1 Objectives of MOD III Modify the GNU parser, *cc1*, such that it outputs the cross-reference information between the *function*, *global variable*, and *typedef* symbols, as done by [4].

3.7.2 Reconstruction of steps for MOD III After initial overview of GCC and its source code, Subjects C and D were told to:

[M3-1] Find the code-generation function called after the parser finishes parsing a function declaration. It would have the parse tree of the whole function. Learn the structure of the parse tree and print it out for some sample programs.

This was based on the hypothesis:

[M3-2] A function is the smallest unit that the parser will pass to the code generator.

```
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.

After two weeks Subject C and D reported that the parse trees they printed did not have information about the whole function. Their experiment was verified by Subject A on the thought that they may have erred. Thereafter, Subject A scanned and read pieces of the parser (written in *yacc* [12]) and various functions in the code-generator. This was a very painstaking exercise that did not provide any new knowledge. The mystery was finally resolved by the following statement found in GCC's documentation:

[M3-3] `The RTL intermediate code for a function is generated as the function is parsed, a statement at a time`

This essentially violated M2-2, our working hypothesis. This revelation and our exposure to GCC code changed our estimate of the task. The exercise was aborted because we found a public domain tool [17] that performed the task we wanted.

3.7.3 Analysis of MOD III

It is not always easy to validate hypothesis or understand design directly from code. We spent a long time trying to map our hypothesis to code or vice-versa. We scanned, read, and traced code. But didn't reach anywhere. Unlike MOD II, the discrepancy between the actual design and the expected one was not easily visible. A possible reason could be that the parsing and code-generation tasks collectively span several files. Getting a bigger picture from voluminous code is not easy. There could be yet another explanation, our *inability to hypothesize the design that was actually implemented*. Our hypothesis, M3-2, was based on text book approach to compiling which is chosen for clarity. The GCC's design is chosen for efficiency. Since none of the subjects were expert in compiler construction we could not think of the GCC's alternative.

3.8 Exercise MOD IV

This exercise differed significantly from the previous ones, as summarized below:

- It was performed on WPIS not GCC.
- It required the development of a lot more new code than MOD I to III.
- The purpose was not to change the behavior of WPIS but to change an algorithm while preserving the behavior.
- The task was in a domain new to Subject E.
- The work was estimated to be significantly large and the chances of success, in Subject A's assessment, very low.

As a result, Subject A decided to maintain strict control on the task and proceed in steps (which coincidentally happen to be the same as Zvegintov's 9–steps [25]). This exercise once again enumerates the use of "mimicking" previous solutions to similar problems. Additionally it observes:

- the difficulty in program comprehension due to "dead code",
- the difficulty in understanding an implementation that differs from the documentation, and
- the need to resort to reading code, when all else fails.

3.8.1 Objectives for MOD IV

Replace the interprocedural slicing algorithm of WPIS, published in [10], by the algorithm in [14].

3.8.2 Reconstruction of steps for MOD IV

Subject E spent almost two months learning about program slicing, learning how to install and use WPIS, and understanding the two algorithms. Subjects A and E discussed possible changes that may need to done in the original code to move to the new code. Subject A gave Subject E a tour of the source code directory inferring what the files implemented from their names.

The source code for program slicing were correctly assumed to be in the directory `src/algorithms/slice`. Interprocedural slicing is essentially a graph traversal problem. The two algorithms of concern differed in how they traversed the program dependence graph, a special type of graph. Subject E was asked to:

[M4-1] Understand how the old algorithm is implemented. See how it operates on the nodes and edges of the program dependence graph. We can "mimic" it's approach for our problem.

Subject E did this task independently. He reported that there were some files that did not make sense; they appeared to be syntactically incorrect.

[M4-2] We found from the author of WPIS that it was dead-code – remains from previous versions of the system.

We could not find that the files were not being compiled because a) we assumed that all the files were compilable and b) the compilation procedure for WPIS is spread over a collection of Makefiles and hence were too complex to verify. In another instance, Subject E reported that the code was not doing what the documentation [10] described. It turned out that:

[M4-3] The traversal mechanism in WPIS' implementation was significantly different from that in the published work [10].

We spent a few hours trying to map the published algorithm to the code and when we couldn't, we inferred that:

[M4-4] Since the program works our assumptions about it must be wrong. We then resorted to reading and tracing through the code.

Once the discrepancy was realized, the modification and implementation was smooth. It actually took less time than originally estimated.

3.8.3 Analysis of MOD IV M4-1 reiterates the tendency to "mimic" existing solutions to subproblems.

Deadcode increases the difficulty in program comprehension. The reasons are obvious. Any code in the program can not be ignored. This is especially so if the code contains "beacons" that are relevant for the task at hand. If the code is obsolete but contains indicators as though it may be relevant to the modification task, a new programmer has to reestablish that it is dead. When the program being studied works, the first reaction to any perception of discrepancy is to consider that you may not have understood it right (M4-4). In the case cited in M4-2, the obsolete code was leftover from an earlier version of WPIS that implemented a different slicing algorithm. We could not believe that the code was syntactically incorrect, after all the system compiled, linked, and executed well. As a consequence we thought may be we did not know something about the programming language C that the author of WPIS knew. Since the compilation procedure was too intricate we did not attempt at verifying if the source code file in question was used to generate the program. Such diversions can be very costly.

It is very difficult to detect that the documentation and implementation differ. In Brooks' terms [3], to understand a program is to create mappings between levels of domains. Given a documentation and an implementation, one may assume that they are mappable and attempt to create mappings between the two. This is futile (and frustrating) when the document and implementation are not consistent. The realization of the inconsistency may take quite sometime.

4 Comparative analyses

The previous section gave a narrative of four modification exercises performed on GCC and on WPIS, systems described earlier. It also presented analyses of each of the exercises. In this section we compare the experiences of modifying GCC with that of modifying WPIS. We also compare the differences between the approach used by Subject A (the author) and Subjects B to E (the graduate students).

4.1 Comparison between systems

MOD I, II, and III were performed on GCC and MOD IV on WPIS. MOD I, II, and IV were successful and MOD III was aborted. MOD I and II required understanding very small portions of GCC and MOD III required understanding a very large part. MOD IV's need to understand WPIS was significant but less than that required for MOD IV. The three modifications on GCC were frustrating. The last one aborted because we felt it was very hard to understand GCC. On the other hand, MOD IV was a pleasure. Because of its success we have undertaken another project, referred to as MOD V, that would affect more than 50% of WPIS' modules. In spite of the magnitude of the changes to be made, we are confident of its success.

Our experience with the two systems can be compared on the following factors:

- our knowledge about the problem domain,

```
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

- organization of source code in files and directories, and
- levels of abstractions in design.

DOMAIN KNOWLEDGE. One of the reasons MOD IV was successful and MOD V is progressing smoothly is that we are conversant with the problem that it addresses, related design issues, the algorithms and data structures. These are all available as published literature that we have been pursuing for the last three years. The subjects were required to spend considerable time reading the literature and understanding the change request, before even looking at the code. In contrast, our knowledge about GCC's internals was negligible when we undertook the exercise. Most of GCC's design decisions are geared towards efficiency of compilation. Hence its design is significantly different from that proposed in traditional compiler construction textbooks. While there is some documentation available on GCC, we did not spend enough time reading it.

ORGANIZATION OF SOURCE CODE. 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 a directory called *packages*. The directory *pdg_module* defines objects in the problem domain using the basic mathematical objects. This module is kept separate, along with other modules defining problem-domain level concepts.

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 no simple task.

LEVELS OF ABSTRACTIONS. Compare the code segments in Figures 2 and 3. They present two functions, one each from GCC and WPIS. These functions were located as candidates for processing two change requests. 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 to tasks in the problem domain. 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.

GCC's decomposition of functions is too coarse grained and completely lacks abstraction of data. Its functions rarely span less than 50 lines and they are usually very intricate. WPIS's functions rarely span more than 50 lines and rarely contain lots of detail.

4.2 Comparison between subjects The author and the students differed considerably in their ability to understand and modify programs. The differences may be attributed to the following obvious factors:

- experience with reading other's code,
- programming knowledge, and
- domain knowledge.

These factors may be further refined into the following sub-factors that contribute to the differences:

- the ability to create hypotheses about the system,
- the ability to make logical arguments to prove or refute the hypotheses based on given facts, and
- the ability to infer a program's behavior from sample inputs and outputs (like concept identification [13, Chapter 7]).

The students were less experienced and also lacked programming and domain knowledge necessary for the work. Their performance was, therefore, poorer but could be improved by training.

5 Conclusions

We have presented a narrative of four program modification exercises performed on two software systems about 50,000 lines in size. The exercises were not performed to study programmer behavior and the narrative has been reconstructed after the fact. Three of the program modification exercise were successful. One exercise was aborted because we found a public domain tool that satisfied our needs. We make several observations from analysing these exercises. These observations, of course, should only be used to create hypotheses about programmer behavior. Unless emprically validated these hyptheses can not (and should not) be taken as theory of programmer behavior.

The observations we make may be summarized as follows. The 9-step process of software change proposed by Zvegintov [25] seem to be a good first level decomposition of tasks involved in modifying a program to "add" features. Modifications made to remove functions, correct behavior, or for complete migration of code are not captured by this model.

The process of understanding programs can be modelled by the hypothesis-test-refine theory proposed by Brooks [3]. The bottom up process proposed by Shneiderman is used in extreme situations [18]. There are circumstances when a programmer just scans code in search of "beacons" to seed a hypothesis. He may resort to reading code or tracing the data flows after shortlisting a set of functions as candidates for change. That a symbol or a code segment may act as a "beacon" depends on the relevance of the information to the specific modification task; it is not an intrinsic property of the code itself.

While documentation and source code are the obvious sources of information, programs are also understood by observing their execution. A program may be executed and its inputs and outputs analysed to determine its behavior. This is done when the documentation is absent or incomplete. Programs may also be executed to get a perspective different from the documentation. This process of understanding a program from its inputs and outputs is analogous to concept identification - for instance, given a sequence of numbers identify the mathematical function used to generate it [13].

The ease with which a program may be understood depends on several factors. It depends on the programmer's experience with modifying code and also his knowledge of the domain and the relevant programming concepts. It is influenced by the programmer's ability to create hypotheses about a program's design (or behavior) and the ability to test the hypothesis using observed facts and logical reasoning. A system decomposed into several levels of abstractions is easier to understand than one with a more coarse grained decomposition. Organizing the source code in a directory hierarchy that reflects this decomposition can further help in comprehension of a large system.

When introducing new features programmers attempt to find subproblems whose solutions are also needed to implement other features of the system. If such subproblems are found, their solutions are mimicked in the code introduced. In a system whose decomposition is too coarse, mimicking a solution may mean replicating large pieces of code. In a system with layers of abstraction, it may simply mean calling the same procedures or functions. Changes such as deleting a feature may make some code obsolete. The obsolete code is often not destroyed and hampers subsequent modification activities.

Bibliography

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

[2] R. Brooks. Studying programmer behavior experimentally: The problems of proper methodology. *Commun. ACM*, 23(23):207–213, Apr. 1980.

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

[4] Y.-F. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Trans. Softw. Eng.*, 16(3):325–334, Mar. 1990.

[5] S. C. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, pages 66–71, Jan. 1990.

[6] K. A. Ericsson and H. A. Simon. *Protocol Analysis: Verbal Reports as Data*. MIT Press, Cambridge, MA, 1984.

[7] R. K. Fjelstad and W. T. Hamlen. Application program maintenance study – report to our respondents. In G. Parikh and N. Zvegintzov, editors, *Tutorial on software maintenance*. IEEE Computer Society Press, 1983.

[8] D. J. Gilmore. Methodological issues in the study of programming. In J. M. Hoc, T. Green, R. Samurcay, and D. Gilmore, editors, *Psychology of Programming*, pages 83–98. Academic Press, 1990.

[9] J. M. Hoc, T. Green, R. Samurcay, and D. Gilmore, editors. *Psychology of Programming*. Academic Press, 1990.

[10] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst.*, 12(1):26–60, 1990.

[11] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Trans. Softw. Eng.*, pages 749–757, Aug. 1985.

[12] S. C. Johnson. YACC - yet another compiler-compiler. Technical report, Bell Laboratory, Computer Science Technical Report, 1975.

[13] W. Kintsch. *Memory and cognition*. John Wiley and Sons, 2nd edition, 1977.

[14] A. Lakhotia. Improved interprocedural slicing algorithm. Technical Report CACS-TR-92-5-8, University of Southwestern Louisiana, Lafayette, Nov. 1992.

[15] A. Lakhotia, S. Mohan, and P. Poolkasem. On evaluating the goodness of architecture recovery techniques. Technical Report CACS-TR-92-5-4, University of Southwestern Louisiana, Lafayette, Sept. 1992.

[16] H. A. Müller and J. S. Uhl. Composing subsystem structures using (K,2)–partite graphs. *Proceedings of the Conference on Software Maintenance*, pages 12–19, Nov. 1990.

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

[18] 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.

[19] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. Softw. Eng.*, SE–10(5):595–609, Sept. 1984.

[20] E. Soloway and S. Iyengar, editors. *Empirical studies of programmers*. Ablex, Norwood, NJ, 1986.

[21] 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.

[22] R. Wachtel. Software analysis notes: series of 22 articles. *Software maintenance news*, 5-7, 1987-89.

[23] R. Wachtel. Software change: a guide for process improvement and automation. Personal notes, Available from: Box 38, Occidental CA 95465, 1992.

[24] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, 1984.

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