APPLICATION OF A KNOWLEDGE ELICITATION METHOD TO SOFTWARE DEBUGGING EXPERTISE

T.R. Riedl
J.S. Weitzenfeld
G. A. Klein
J.T. Freeman

ABSTRACT Critical Decision Method (CDM), a knowledge-elicitation technique, was utilized to investigate C language debugging in Bell Laboratories as part of the design of a new course. A total of fifteen informants who were designated as expert C debuggers by their management, were drawn from a variety of projects and organizations and were interviewed using CDM. Eight of these experts subsequently participated in a panel session intended to validate and clarify the data. This paper reports the major findings, the most striking of which were the important role played by various kinds of interpersonal skill and the role of metacognitive skills. Another key finding, consistent with the research literature, concerns the critical role played by strategy and planning in determining expert debugging behavior. Contrastingly, less expert debuggers generally engage in more random and less efficient debugging activity. It was also found that maintenance problems were the type which experts spent most of their time debugging. A number of specific techniques and relevant skills characterizing expert debugging are included.

THE CONTEXT

The present study is part of a needs assessment and course design project in software engineering. An earlier survey (Riedl, 1988) had established the need for a course on debugging skills. That survey showed a perceived need for a course teaching the use of debugging tools and specific "tricks" used by expert debuggers. However, there already are KETC courses in debugging tools, and we suspected that there was more to being an expert debugger than knowing a bundle of tricks.

In order to discover what knowledge or skill gaps might affect actual debugging performance in Bell Laboratories environments, we used the Critical Decision Method of knowledge elicitation with a total of 15 expert C debuggers, drawn from a variety of projects and organizations. This was followed by a subsequent panel of 8 of the subject matter experts (SMEs) (Weitzenfeld, Freeman, Riedl, & Klein, 1990). A report of the findings was circulated to the participants, together with a brief questionnaire asking respondents to rate how often certain techniques were used, how efficient it was when used, and the room for improvement among debuggers on the method. The results of this poll were used to select methods for more intensive consideration in the meeting. The present document presents only some of the conclusions.

INTERPERSONAL SKILLS

The first striking finding was that, at least in these incidents selected by the informants as challenging, there was a large amount of social interaction. All of the debugging episodes mentioned involved more than one person, either as a team effort, as sources of information, or as someone with whom to discuss ideas. These interactions required social, political, and interviewing skills. There was some testimony from the informants that these skills improve over time and that they can be taught. The social skills involved coordinating and leading groups, using others to help clarify or to redirect one's own thinking, and networking to establish contacts who could answer questions about areas beyond the experts own expertise. We were told that it helped to bounce ideas off someone. One area of expertise was the selection of a team, for which differences in personality and working style should be considered as well as specialized knowledge. The work is more efficient if the team interacts spontaneously.

It is important to have contacts or to know who would have contacts in specific areas that might affect your work. One respondent mentioned the use of design reviews as networking opportunities because you can not only establish ties, but also get a reading on the other person.
Another important role of the design review experience in developing expertise was as providing a model of "the other" to, as one developer put it, "keep you honest." We were told that there is a general tendency to let some difficult steps slip, which were easier to force oneself to do when one had to give an account to someone else. This is similar to, but different from, the brainstorming function of talking a problem over with another programmer. When someone else is not available, one debugger told us it helps "to make a presentation to the wall," although the wall is not as effective as another person. Political and negotiating skills were involved in debugging when it was not clear where a bug lay, and, consequently, who owned it. Informants referred to bringing "ammo" to meetings, bringing "my own code so it wouldn't look as if I were accusing," and the recurring problem of getting someone to look at their code with your problem in mind. Although cooperation typically was forthcoming, it was not a matter of course: there was one episode in which an informant went to a meeting with another MTS to localize a problem and found the other's supervisor there as well. As the quotations above indicate, we heard different approaches to negotiating (em the accusatory and the non-accusatory, reflecting the appropriateness of different styles to different situations. It appeared to be easier to get cooperation at one of our sites than at another. It was clear that many of our respondents had to plan how to get sufficient cooperation from members of other groups. A simple request for information about the structure of a module might get a quick response, but it took much more effort for holders of MRs to persuade the person responsible for a block of code that was interacting with the ostensibly buggy module to come to the lab with them.

It is also important that any effort to repair the process that caused or permitted the error would particularly require political skill. Although such efforts went beyond the responsibilities assigned to our respondents, they occasionally could make suggestions. In addition, we were told that there are certain types of people who will only cooperate when their pride is challenged. The ability to assess the personality of domain experts may be an important part of debugging expertise in some environments. Some people, we were told, require some warm-up chatting before a problem is posed; some can be counted on to follow up on a bug they are told about, whereas others will only work on it if an MR is written. Some people are best approached by getting them interested in the problem. Efficiency is improved by knowing how to deal with different kinds of people.

Interviewing skills also were necessary to elicit from non-technical originators of reports just what the relevant conditions were. An interview template or protocol would help ensure that even the obvious information is conveyed, such as the fact that a problem is intermittent. Specialized branching of lines of questioning can be included in an intelligent job aid, or taught to debuggers. An example is the follow-up question "What time of day?" if an intermittent problem occurs every day. One informant told us that a mentee typically came back from an interview with "fuzzier" answers than the SME would have obtained.

The interviewing skills called upon go beyond having a uniform set of questions to ask. For example, a bit of wisdom for trouble-shooting interviews was the necessity to "validate" the originator's problem, i.e. let the originator know that you understand that (s)he has a significant problem. Otherwise the conversation will get stuck at "This is very bad. This is a serious situation, etc." Other skills are related to the complexity of the environmental conditions that can precipitate a problem. Environmental influences can go beyond the causes included in a standardized questionnaire, and at least one informant told us that it was a good idea to try to find out everything the originator knows about the situation. A particular difficulty arises when 2-3 layers of intermediate field personnel must translate inquiries. A related finding that may be a teachable maxim is that experts rarely read unfamiliar code if they can get someone already familiar with it to tell them about it or to read it for them. The entire research literature on how programmers read unfamiliar code is only minimally relevant if it is more cost-effective to establish interpersonal networks so that code is read by people already familiar with it. Two of our informants spoke with no embarrassment of reading strange code and "not understanding it." One said, "I wasn't going to go into that spaghetti!" In fact, knowing when you don't understand code may be an aspect of expertise. (We shall return to the question of reading code when we talk about specific skills of the expert debuggers. Our point here is that social skills are necessary to get the right person to read the code and interpret it.) We also were given some advice for developers on how to ask for help when one is stuck. One should bring the error output (which is often all that is brought by non-experts), but also a list of what was seen and a list of recent changes. One should know the problem domain, and not come back for more help until one has more information than was
brought the last time. We heard about the "favor bank" - the tendency to give help in exchange for past help received or future anticipated help, which tends to provide most help for those who need it least.

**GENERAL CHARACTERISTICS OF EXPERTS**

The literature on expertise already tells us that experts tend to do relatively more upfront planning of strategy before acting than do nonexperts. This has been observed in a variety of fields, including programming (Chi, Glaser & Farr, 1988). We found this to be true of our experts also. We saw signs of it in the two less expert debuggers we interviewed, and our informants often told us of seeing people leap into action without knowing what they were looking for. Our SMEs generally used the progressive deepening strategy first observed in chess grandmasters by deGroot (1965). They use a variety of methods to get an overview of code and the environment. These include documentation of various kinds, asking people, skimming the code and path analysis. Even when dealing with their own code, they tended to refresh their memory about the logic of the program and to do a high level control flow analysis if they had been away from the program for a while. Methods for determining the flow of control in new code included setting break points, and forward and backward analysis, depending upon the way the problem presented itself.

Experts tend to be aware of a broader range of environmental features, from the hardware on which things are running, including aspects of the configuration (such as shared memory), to the activity of the compilers, to user behavior. Many factors other than software bugs can take on the appearance of bugs, and many bugs only appear when the environment in which the software is running changes. Experts have a rich picture of these factors. Our informants seemed to be keeping a steadily updated and broad model of the problem environment in their heads - one whose relative level of detail in different areas they could assess. This model was used when they asked themselves some version of the question "If I had to write code to make this happen, how would I do it?"

On the basis of this overview, some of our informants were able to estimate both the amount of effort it would take to examine a search path and of the likelihood of finding the fault there. Some informants specifically said that they could quantify the strength of their hunches about where a fault did or did not lie. (Such comments typically came from informants who were not on the giant projects.) These estimates were based on factors such as knowing what is still working right, what is likely to touch the symptomatic point, the known bugginess of certain modules, and previous experience with the code (producing either confidence or lack of confidence in its quality). These hunches were based partially on rules that could be stated, such as "If it happens on initialization, immediately look at the code that does the initialization," but also on recognitional capacities. For example, "this pattern looks like the number table." We shall return to recognitional capacities when we discuss skills, below.

Once our informants have a mental list of possible causes, they work to prune the list and localize the fault by eliminating lines, paths, scripts, times, etc. There are specific strategies and tools used in doing this. Experts classified some problems as "intermittent," and usually focused on how to reproduce these problems. One of the implications of a problem's being intermittent was that its cause may not be in your code, but in environmental changes. It was important to make this global localization early in the process. Our informants talk about this as "reproducing the problem," but, of course, in a logical sense, when you can turn a fault on and off you have localized the problem. A solution to the problem is merely being able to turn off the problem without other untoward effects. If a situation can be made functional by work-arounds or undoing the symptoms, but the original problem not reproduced at will, it is a sign that there still is an error around.

An interesting strategic rule was to check out the easy search paths first, not the most likely. Apparently, the difference in effort between doing the easy things and doing some of the hard things was so great that it made sense to run through the easy search paths to eliminate the possibility of an unnecessary search of the difficult paths. Once we observed this strategy, it was easy to work out a decision tree that would justify it. However, an additional factor appears to be that code is read differently when the developer is certain that it has a bug. By ruling out other possibilities first, our informants justified a more intensive reading of the difficult search paths. Debugging experts check out easy possibilities even when they are reasonably confident that the problem is not located there. What is easy to do, however, depends significantly upon the
resources available. On some projects appointments must be made to use the lab, or it will take 2-3 hours to set up a model. These variations in access to resources and in how easy it was to test things affected the appearance of the search pattern.

Our informants tended to use debugging tools in the process of listing or eliminating alternative possibilities, not as exploratory devices. They almost always are looking for something specific and several informants contrasted their slowness to use tools with the scattergun approach of others. Paradigm examples of tool use included using DART to verify that a table was good before a function and bad afterwards, or using C-scope to trace the functions that touched a data structure. Sometimes the output of a tool, such as a protocol analyzer, was scanned for "anything that could kill a call." This presumes recognitional capacities that, we were told, people new to the project would not have.

Since our sample of episodes was relatively small and biased toward the exceptionally difficult, we do not want to make generalizations about which tools were most useful. However, in addition to recognized debugging tools, we did find more than one of our experts using an editor as a debugging tool. One, in particular, used a 150 line terminal and editing functions for search and compare operations, with the possibility of keeping several windows open at once. He searched for all calls to the next phase of a program, then searched backward, pinching off a fourth to a third of the branches as unlikely as he went. The advantage of this less automated mapping of the program logic was the programmer's ability to use likelihood judgments to simplify the map. The reduced tree was a more useful representation of the programmer's knowledge state than an exhaustive tree.

**SPECIFIC TECHNIQUES**

We found several techniques that could be of use in specific relevant situations. (Detailed results are omitted because of space limitations) These included: center, tab (#); lw(2.5i) lw(2.5i). T{ The use of timelines Lists of interfaces Lists of messages sent and received - together with the accompanying load, tools used to communicate and whether a response was expected or not Forward analysis Backward analysis Keeping a textfile of assumptions Truth tables - done in the head or on paper T}#T} Statistical modeling of hypothesized faults (deducing the probability of various symptoms from a model of the fault and the various loads) Turning breaks on and off for an optimal information delivery Putting in "verbosity" - messages, such as time stamps Manipulating variables and checking to see how printouts changed. T}

Our informants viewed reports, including error messages, as machine behavior, or symptoms to be explained, not as true statements. That is, they tended to be aware of the causal chain required to produce the message they got or didn't get, and to treat that as information about the state of the system. They were aware of the factors that keep error messages from printing, and that error messages can be caused by factors other than those intended by the authors of the messages. They also drew inferences from the rate of error messages and from the number of different messages sent. The general procedure for debugging from an error message was to work backwards along the causal chain from the message.

**SKILLS**

We found certain skills and dispositions in our experts, some of which probably develop over years of experience and some of which are most likely enduring personal characteristics that lead one to become an expert debugger. It is an open question to what extent these can be accelerated or developed by education and training.

To begin with some general ones, our experts thought about the impact of the system on AT&T's customers. They did not like to leave unsolved problems and, after locating a problem, they often asked themselves if there were likely to be similar problems around, if one of the alternative fixes would have less impact on customers, if they could leave warnings of potential future problems where they would be seen, etc. They tend to use all the cues available, and not overlook "trivial" signs that things are not quite right.

Several of our informants said that they had learned from experience that the problem always could be in their code. However, in the course of debugging they encountered others whose first reaction was that "The
problem couldn't be in my code." Our informants occasionally used accusations that the bug was in another group's code as negotiating ploys to get specialized help they needed, but this was done tactically and not as a matter of course. If anything, we found that the expert debuggers were reluctant to give up ownership of a problem because they were more sure it would be resolved properly if they kept it. On a more cognitive level, we received indications that there are many different ways to read code, and our informants tended to be aware of how they were reading code. There is skimming it to discover the logic. There is a line by line reading. There is searching it with a hypothesis in mind. There is reading it with a problem in mind, e.g. "I know what the output is and I look for something that will make it do that." Several informants said that 5-6 years of experience on a project gives one a different sense of what it means to read code carefully. More than mere knowledge of that project, it also allows the expert to be aware when reading code in a new project of the detail that is being missed. We heard expressions such as "Getting my mind in a state to interpret codes," and "I read it trying to understand every line." There are suggestions that such close reading means inferring the file structure, etc. It was suggested to us that many developers don't know what it is to read code carefully. This may be one reason why good debuggers tend to get experts on a specific body of code to read it when a search is necessary. One informant mentioned that possessing a good memory for variable names, so that it is easier to skim code looking for changes in variables. This may be the kind of "skilled memory" referred to in the cognitive psychology literature (Chase and Ericsson, 1981,1982; Staszewski, 1988). Skilled memory is learned capacity to handle greater quantities of information in short-term memory within a specific domain. It generally develops over two to three years of practice. For example, lightning calculators develop such a memory for numbers. Such specialized memory would make it easier, and in some cases possible, to read new code and understand what is happening. Our panel members generally agreed that they could assimilate new code faster than less expert debuggers.

One informant mentioned always running over in his mind his assumptions and what he thought he knew about the problem. Every time a new piece of information came in, he reviewed his list. Some people keep such lists of assumptions on paper, but, according to this informant, those who get to be good debuggers keep them in their head from the very beginning. From the point of view of cognitive psychology, keeping the list on paper would remove some memory load. However, it clearly is easier to manipulate assumptions if they are in your head. Another skilled memory domain may well be the ability to keep a problem representation in mind. If so, it would make sense that those who get good keep their representations in their head from the start. It would be harder for them, but it would promote the development of the skilled memory.

**FUTURE CHALLENGES**

As systems become more complex and subject to increases in load and demand, new problems are being created while previously dormant ones are surfacing. Our data reveals something about the nature of maintenance debugging problems, as well as the expertise required to solve them. The very nature of emerging hi-technology may require the development of new skills and expertise. Thus, there will continue to be a need for work of the kind presented here. If properly carried out, continued research in this area will result in technology that reflects the greater levels of expertise involved in creating it.

**REFERENCES**


