Interactive Explanation of Software Systems

W. Lewis Johnson and Ali Erdem
USC / Information Sciences Institute & Computer Science Dept.
4676 Admiralty Way, Marina del Rey, CA 90292-6695
WWW: http://www.isi.edu/isd/johnson.html, erdem.html
{johnson,erdem}@isi.edu

Submitted to the 10th Knowledge-Based Software Engineering Conference (KBSE-95)

Abstract

This paper describes an effort to provide automated support for the interactive inquiry and explanation process that is at the heart of software understanding. A hypermedia tool called I-Doc allows software engineers to post queries about a software system, and generates focused explanations in response. These explanations are *task-oriented*, i.e., they are sensitive to the software engineering task being performed by the user that led to the query. Task orientation leads to more effective explanations, and is particularly helpful for understanding large software systems. Empirical studies of inquiry episodes were conducted which support this claim. The I-Doc tool is being developed to embody these principles, employing knowledge-based techniques. The presentation mechanism employs World Wide Web technology, making it suitable for widespread use.

1 Introduction and Motivation

Software engineers, and software maintainers in particular, spend significant amounts of time attempting to understand software artifacts [2]. These software understanding activities have been characterized by Brooks and Soloway [1, 15] as being composed of inquiry episodes. According to Soloway et al., inquiry episodes involve the following steps: *read* some code, ask a *question* about the code, *conjecture* an answer, and *search* the documentation and code for confirmation of the conjecture. Because of the important roles that conjecture and search play in the process, Selfridge has described software understanding as a discovery process [14].

Search in software understanding is very error-prone; people do not always know where to look for information to support their conjectures. In Soloway's studies the most successful subjects systematically scanned code and documentation from beginning to end, to make sure they found the information they required. This is clearly impractical for large systems.

The solution that Soloway and others advocate is to organize documentation to make search easier. Two principal methods have been attempted. One is to *link* documentation, so that the understander can easily get from the site in the documents where the question is posed to the site where the answer can be found. Soloway uses this technique to document delocalized plans, linking the various elements of the plan together. The other method is to *layer* documentation, so that different types of information reside in different layers. For example, Rajlich [12] organizes information into a problem domain layer, an algorithm layer, and a representation layer. Understanders can then limit their reading and searching to particular layers. The conjecture-and-search method of obtaining answers to questions is essentially unchanged in these approaches, but the search process is made more efficient.

Of course there is another common technique for obtaining answers to inquiries about software—to ask somebody who knows. There is no need for searching at all with this method. Our objective is to develop a tool that emulates the ideal of having an expert on hand to answer questions. Such a tool should be able to respond directly to the user's inquiry with information that helps provide an answer.

Research in automating consultative dialogs has identified a number of important requirements for explanation systems [11]. First, they must of course have the necessary knowledge to provide the desired answers. Second, they must provide answers in a form that the questioner can understand, and avoid concepts that the questioner is unfamiliar with. Third, they should take into account the goals of the questioner. The content of the answer can depend upon what the user is trying to do with the information.

Principles similar to these are already the basis for the design of certain types of user manuals, namely minimal manuals [9]. Such manuals attempt to anticipate the tasks that users might need to perform, and provide information to help achieve them. Although advocates of minimal manuals claim that novice users are in particular need of task-oriented documentation, it is reasonable to hypothesize that software professionals would benefit as well. Lakhotia [8], for example, quotes a software developer who says that what he would like in the way of a software understanding tool is "something that helps me get the job done, fast." Unfortunately, the tasks of software professionals in general are not precisely defined. User manuals can be oriented toward specific tasks, such as composing a letter or generating a mailing list. Software engineering tasks such as design or maintenance are much broader, and the influence of such tasks on software understanding is unclear, although software understanding researchers such as Brooks have conjectured that such influences exist [1].

In order to develop tools that approach the ideal of an on-line software consultant, our work has proceeded on two thrusts. First, we have examined the interactions that occur during expert consultations, in order to determine what types of questions people ask and what governs the answers that the experts provide. We have been particularly interested in the following.

- What information about the questioner (e.g., task, level of expertise), determines the form of the answer?
- What kinds of information do people ask for? To what extent is it available from conventional documentation sources?

We are using these results as the basis for the design of a new kind of an on-line documentation tool. Information provided by the tool is explicitly organized in the form of answers to common questions. Like most expert consulting systems, and unlike typical on-line documentation, the system has an explicit model of the user and his or her task. This enables the system to select information that is likely to be useful, and present it in an appropriate form. The objective is to increase the likelihood that the information provided answers the user's question, and to reduce the amount of search and interpretation required of the user. The presentation medium being employed is *dynamic hypermedia*, i.e., hypermedia descriptions that are created dynamically in response to user queries. Automated text generation techniques can make hypertext a medium for human-computer dialog, with features similar to that of interactive question answering systems. The hypermedia descriptions are presented using commonly available tools, namely World Wide Web clients.

2 Empirical Study

2.1 Motivation

In order to find the answers to the questions above, we decided to analyze the messages in Usenet newsgroups. We were particularly interested in the hierarchy of newsgroups under comp.lang, which contain discussions about programming languages. The articles posted to comp.lang newsgroups included a wide range of questions about programming languages and answers to them. The dialog between the user and the advisor was clearly observable from these messages. The variety in users' backgrounds, expertise and activities made these newsgroups a perfect place for studying software inquiries.

We focused our attention on the comp.lang.tcl newsgroup, which contains discussions about Tcl and Tk programming languages. Tcl is a simple textual language and a library package. Its ease of use and simplicity makes it useful for writing shell scripts and prototyping applications. Tk is an extension to Tcl and provides the programmer with an interface to the X11 windowing system. When these tools are used together, it is possible to develop GUI applications quickly. However, because of the limitations of the documentation and frequent upgrades to these products, some users rely on the newsgroup to get answers to their questions. Although it is possible to answer some of the questions by consulting the source code, it requires good knowledge of C and the skills to find out the relevant sections in thousands of lines of code. The source code is well documented, but it is fairly large, comprising a total of over 100,000 lines of C code. We used the information available in documentation and source code for identifying user's familiarity with these languages and determining how easy it was to find out the answer in the documentation.

2.2 Data Profile

Since most of the users in the newsgroup were programmers, the data was biased and the number of questions about different user tasks were not equal. For example, there were less application users than programmers. Similarly the number of maintenance programmers was much lower than program developers. However this difference is not as significant as the previous one, since both Tcl and Tk are interpretive languages. Egan claimed that there is some overlap in terms of the mental processes for coding and debugging in interpretive languages [4]. Programmers using interpreted languages generate a small amount of code, read it for comprehension and correct errors in a continuous fashion [4]. Although most of the questions posted to this newsgroup came from program developers, we believe that similar questions will be asked by maintenance programmers when they try to understand the same programs.

2.3 Data Analysis

We read 1250 messages posted to the newsgroup between 2/17/95 and 4/22/95. For data analysis we followed a method similar to [5] in that we considered only the messages that asked questions about Tcl/Tk. Messages asking irrelevant questions (distribution sites, FAQ location etc.), product announcements, opinions were ignored. We found 249 questions and classified them as follows:

We first tried to estimate the user's expertise level. In nearly all cases, the expertise level was easily inferred. It was either stated explicitly in the message or was easily guessed by looking at the contents of the message. If the user stated that he just started learning Tcl/Tk or asked a very simple question that was covered in the documentation, we classified him as a novice. If he had been using Tcl/Tk for more than a year or asked complex questions that were not in the documentation, he was classified as an expert. All others were classified as intermediates.

Second, we tried to determine the user's task. We found it useful to characterize tasks at two levels: the *macro-task* and *micro-task* levels. A macro task is an activity that the user performs on the system as a whole, e.g., maintaining it. A micro task is a more local activity performed on a specific system component, e.g., forking a process, configuring a widget, or invoking the `make` utility. Macro tasks were categorized as follows:

- Installer: Users who are installing or upgrading Tcl/Tk
- User: Users of Tcl/Tk applications
- Integration programmer: Programmers who are trying to integrate Tcl/Tk with C by calling Tcl/Tk functions from C or vice versa
- GUI programmer: Programmers who focus on graphical user interface issues
- Communication programmer: Programmers who develop applications that communicate with other applications running on the same or a remote computer

- Other programmers: All other programmers including UNIX shell programmers

After the user's expertise level and task were determined, we looked at the type of the question. User questions were either *goal, problem* or *system oriented.*

- *Goal Oriented*: These questions requested help to achieve task-specific goals and were further categorized as follows:
  - *Procedural*: Questions like *How can I read a file into an array?* asked for a plan to achieve the goal.
  - *Feature identification*: The questions in this group were like *Is it possible to display a picture on a button widget?*. They differed from procedural ones, because the user was not sure whether the goal was achievable. However, usually the answers to both types of questions included the plans to achieve the goal.
- *Problem Oriented*: When the users failed to identify the source of a problem, they asked this type of questions. An example was *Tcl installation fails with an error message. What am I doing wrong?*
- *System Oriented*: These questions requested information for identification of system objects and functions.They consisted of:
  - *Motivational*: The users tried to understand why the system functioned in a particular way and how that behavior could be useful. An example question was *Why is the ability to bind to the CreateNotify and DestroyNotify events not supported in Tk bind command?*.
  - *Conceptual*: These questions asked for descriptions of system objects and functions. An example was *What is an option menu?*.
  - *Explanatory*: These questions requested explanations about how the system worked. An example was *How does auto_path variable work?*.

Wright claimed that users' questions are either task oriented or symptom oriented [17]. This is the same as our goal oriented and problem oriented classification. System oriented questions are less frequently asked than others, but they are important for some users. Motivational and conceptual questions are important for novices and explanatory questions are frequently asked by experts.

In addition to these categorizations, we also noted whether the message contained any task descriptions and if they were general or specific descriptions. Code samples and error messages were classified as specific, descriptions of desired outcome with no specific information were classified as general.

Finally we identified the target for each question in order to find out the relations between question type, level of expertise and target. As in Herbsleb and Kuwana's study [5], we defined target as the subject of the question or the task user was asking about. For example *How can I pass an array from Tcl to C?* had the target array.

2.4 Results

After all the messages were classified, we counted the number of messages in each group. Figure 1 summarizes the distribution of questions by macro task, expertise and question types.

2.4.1 What kinds of information do people ask for? How does user task and expertise influence the question?

The type of question users ask is predictable to a certain extent if users' task and expertise level is known. For example, installers were more likely to ask problem oriented questions than others. This might be due to the fact that most installers were new users and didn't know enough to identify the problems. Besides, some of the

Macro Tasks	*Goal Oriented*						*Problem*			*System Oriented*									Task Total
	Procedural			Feature			Problem			Motivational			Conceptual			Explanatory			
	N	I	E	N	I	E	N	I	E	N	I	E	N	I	E	N	I	E	
Installer							7	2											9
User				2	2					2			1						7
Integrator	5	4			2		1	6										1	19
GUI Prog.	11	65	5	5	34	1	4	25			5		1	1		1	3		161
Comm. Prog.		3	1					2									1		7
Other Prog.	5	14		2	5	1	5	7						1		3	3		46
Total	21	86	6	9	43	2	17	42		2	5		2	2		4	7	1	249
Type Total (%)	113 (45%)			54 (22%)			59 (24%)			7 (2%)			4 (2%)			12 (5%)			
Section Total	167 (67%)						59 (24%)			23 (9%)									

Figure 1: The distribution of messages by task, expertise and question type (N: Novice I: Intermediate E: Expert)

installation problems were complex and required extensive knowledge outside user domain, like UNIX operating system, libraries etc.

Task by itself was not the only determiner of question type. Expertise level was also important. Figure 2 shows the percentage distributions of question types by expertise level. It can be seen that more conceptual and motivational questions were asked by novices. As users became more familiar with the system, they asked less questions and they were more likely to be goal oriented. Problem oriented questions became less frequent, since a problem identification repository was built during the learning process. System oriented questions decreased from 15% for novices to 8% for intermediates. Users' knowledge about system objects at this level was high enough to reduce the number of conceptual and motivational questions, but was not high enough for asking more explanatory questions. For experts, system oriented questions increased to 11% mainly because of explanatory questions.

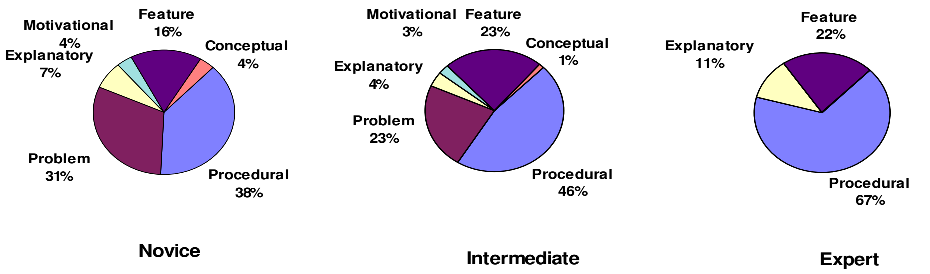

Figure 2: Distribution of question types for different expertise levels

Hill and Miller studied the types of questions asked by users of an experimental graphical statistics system [6]. They categorized questions differently like plans to subgoal, describe system object etc. When we converted their results to our categorization we found that in their study goal oriented questions were 70%, system oriented questions were 22% and problem oriented questions were 4% of the total questions. In our study, problem oriented questions were more common (24%) probably due to the nature of the programming activity, but goal oriented (67%) questions were asked as much.

2.4.2 What information about the questioner determines the form of the answer?

Users' task and expertise level were inferable from the message and this information affected the form of the answer. Sometimes novice users only got the pointers to the documentation whereas experts usually received more detailed and explanatory answers.

Although experts ask less questions, these questions are more complex and harder to answer. For example, a novice procedural question like *How can I display bitmaps on a button?* is easier to answer than an expert procedural question like *How can I scroll the text in a widget during text selection?*. We haven't attempted to measure the complexity of the targets, but attributes of the target like complexity, generality affects both presentation (e.g. present simple concepts before complex ones) and content (e.g. do not present complex concepts to novice users) of the documentation.

The way the questions were asked, possibly because of individual differences, also affected the form of the answer. Some users requested brief information whereas some others wanted detailed answers with explanations. For example in the following message, the user was not only interested in identifying the problem but also wanted to learn how open worked.

I am trying to talk to a process by opening a pipe as described in the Tcl book (set f [open —prog r+])

prog, however, wants its input in stdin only - so it exits complaining...

What does really happen in 'open'? Is there any way out of this?

Soloway et al. found that users employed two macro strategies for finding the answer to their questions, systematic and as-needed [15]. Systematic strategy users read all the documentation whereas as-needed strategy users sought information only when necessary. Research in behavioral theory supports this observation. It is known that when faced with a problem some people use just enough information to arrive at a feasible solution (satisficers) whereas some others gather as much information as they can (maximizers) [13]. Individual differences has to be taken into consideration in answering users' questions.

2.4.3 To what extent is the information available from conventional documentation sources?

Half of the questions could have been answered by consulting the documentation or source code. However, Tcl/Tk experience and expertise was necessary to answer the other half. A simple looking question like *Is it possible to do multitasking in Tcl?* required extensive Tcl and programming knowledge.

Searching the source code was easier if the program code that implemented the answer was localized. The answer to the question *How can I put an image on a button?* was easier to find than *Is it possible to deactivate the toplevel window until another event?*, because Tk code was structured around graphical objects. The documentation, which was structured similarly, was maintainable, but didn't make finding the answer to the second question easier. Documentation that supported delocalized plans could have shortened the time to find the answer [15]. However, the information that needs to be delocalized depends on the task and separate documentation is required for different tasks, e.g. programmer's manual, maintenance manual etc. It is easier to maintain a question answering system's repository than task dependent manuals.

It was impossible to find the answers to certain questions in documentation, since they were either asking for high level plans or instances of a general plan. A question like *How can I split canvas into pages and print?* asked for a high level plan. The answer to the question *How can I pass an array from Tcl to C?* could be answered easily if one knows that Tcl is a string based language and it is possible to pass these strings to C and do data conversion. Once a person learns this general plan, it is simple to answer questions like *How can I pass data-type from Tcl to C?* Although it is not feasible to include the answer to each data-type specific question in static documentation, it is easy to generate the answers to these questions in a dynamic documentation environment.

2.4.4 Importance of Examples

Examples had an important role in both questioners' and answerers' messages. Figure 3 summarizes the number of task descriptions observed in different types of questions:

Examples were most frequently seen in problem oriented questions (81%). It was the easiest and most descriptive way of describing the error and presenting the solution. Similarly 51% of goal oriented questions included examples. However, the task descriptions in goal oriented questions were more general than problem oriented ones. Especially complex tasks were specified with general descriptions rather than specifics.

Question Type	No example	General description	Specific Description
Procedural	54 (48%)	22 (19%)	37 (33%)
Feature	27 (50%)	16 (30%)	11 (20%)
Problem Oriented	11 (19%)	3 (5%)	45 (76%)
Motivational	2 (29%)	4 (57%)	1 (14%)
Conceptual	3 (75%)		1 (25%)
Explanatory	4 (33%)	2 (17%)	6 (50%)
Goal Oriented	81 (49%)	38 (23%)	48 (28%)
Problem Oriented	11 (19%)	3 (5%)	45 (76%)
System Oriented	9 (39%)	6 (26%)	8 (35%)
Total	101 (41%)	47 (19%)	101 (40%)

Figure 3: Distribution of examples by question type

2.4.5 Importance of Discovery Sharing

Usually the answerer knew the solution, since he experienced the same problem. For example, a person from Germany asked how to display umlauts in entry widgets and not surprisingly the answer also came from Germany. Accumulating user discoveries in a repository will provide learning and discovery sharing capabilities.

2.5 Implications for I-Doc

Some of the properties of good documentation [10] and this study's implications on them are as follows:

- *Organize around users tasks and goals, hide unnecessary complexity:* We will tailor the documentation depending on user's task, expertise and individual characteristics and present the answer in a brief, understandable form. Task orientation is important in preparing the documentation, but support for individual differences are also important and they are going to be included in I-Doc. Details not related to the task and information that is too complex for the user will be filtered out to make documentation easier to understand.

  Dynamically generated documentation can support multiple delocalized plans depending on the task and is easier to maintain than task dependent manuals. Such a system's knowledge representation has to be broad enough to incorporate information from existing documents and support new ones. I-Doc's repository mechanism currently has this capability.

- *Support discovery sharing:* Discovery sharing is an important part of the documentation. Currently it is possible to annotate documents in I-Doc. We are planning to add a central repository of annotations to support discovery sharing.

- *Include examples:* Examples are important for understanding the user's questions and presenting answers. We are going to study examples further and include support for situation specific example generation.

3 I-Doc System Architecture

An initial version of the I-Doc system is up and running. Its capabilities are broadly consistent with the results of the above empirical study, and are being extended to further reflect these results.

The system has three major components: a *software repository*, a *presentation generator*, and a *viewer*. The software repository contains annotated software artifacts, and provides the information necessary for question

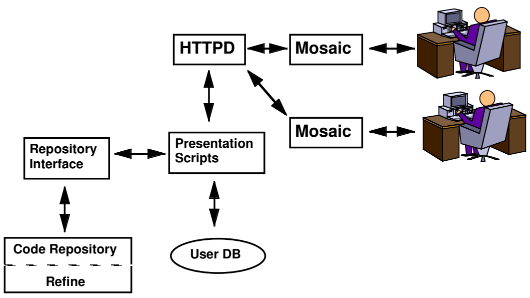

Figure 4: I-Doc System Architecture

answering. It responds to requests from the presentation generator for information relevant to the user's inquiry. The presentation generator takes this information and uses it to compose a presentation description. This is then sent to the viewer, which displays it. The viewer also accepts user queries, which are passed to the presentation generator. Currently this is accomplished by showing the user a list of hypertext links, each of which represents a different query; the user then selects a link from the list. For example, one of the links might be labeled with the string "What are the platform dependencies?". By clicking on this link the user is able to obtain an answer to this question.

Figure 4 shows how the components described above are currently implemented. The software repository is built on top of Software Refinery^{TM}. The repository consists primarily of annotated source code, together with pointers to other available documents. Representations of other types of objects besides source code are also included. The system is currently designed to process Ada code, although there are plans to support other languages. Annotated source code was chosen as the primary information source because it is mechanically analyzable and is usually available for an implemented system. An interface process is used to transmit between the repository and the rest of the I-Doc system.

The presentation generator is built upon HTTPD, a common World Wide Web (WWW) server. HTTPD is usually employed at Web sites to transmit files to remote users. However, one can also configure HTTPD to run programs to generate the information to be displayed, rather than to access a file. We have implemented a number of presentation scripts to be executed in this manner. The scripts access two information sources in order to generate presentations: the software repository, and a database of information about each I-Doc user. HTTPD provides security and password protection, to control access to the software repository.

To access I-Doc, one uses standard Web clients, such as NCSA Mosaic or Netscape. This makes it easy for multiple members of a software project to obtain information about a common system, and reduces the need for special-purpose software. A demonstration version of the system is accessible via the Web addresses at the beginning of the paper.

4 Examples

The following example screen images illustrate how I-Doc works. The system documented here is Advanced Multi-Purpose Support Environment (AMPSE), a system developed by TRW in Ada for the U.S. Air Force. This particular example was constructed from the Ada source code and corresponding design documents, which were annotated and cross-linked by hand. Editing facilities have since been built which enable I-Doc users to create such annotated knowledge bases themselves.

Each I-Doc user must provide some simple profile information: their role on the project (e.g., application programmer, maintainer, etc.), their macro task, and their level of familiarity with the system. In this example, the user has selected Application Programmer as the role, Interface to System as the task, and High as the level of familiarity. In other words, the user has an understanding of what AMPSE does, and is building an

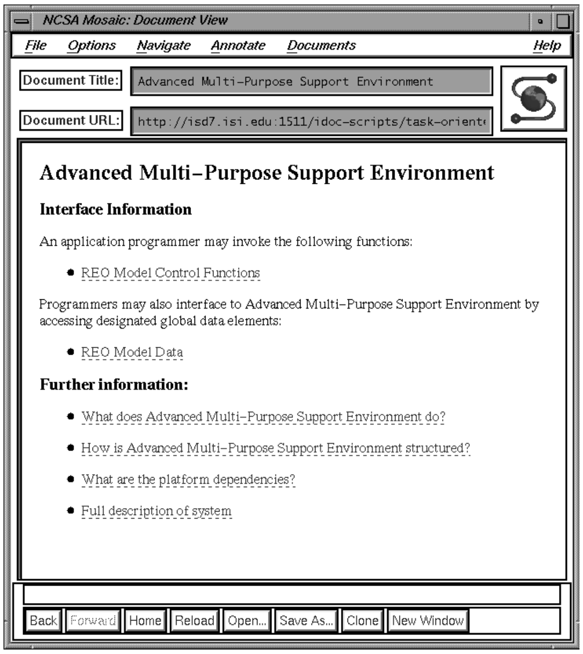

Figure 5: High-level description of the AMPSE system

application which interfaces with it.

Figure 5 shows an overview of the system, from the designated perspective. The description is a combination of dynamically generated structured text and dynamically constructed hyperlinks. The objective is always to provide a manageable amount of information relevant to the user's question. Two types of hyperlinks are shown: links for obtaining elaborations on the information presented, and links for obtaining answers to other questions. An example elaboration link is the link labeled "REO Model Control Functions", which provides information about procedures that external applications are expected to invoke. An example question link is the one labeled "What are the platform dependencies?" When this link is selected, available information about platform dependencies is provided. Such links are needed in case information that the presentation generator filtered out is in fact of interest to the user.

Figure 6 shows what results if the information is not filtered using the user model. All attributes of the system are then shown, only some of which fit on the screen shown.

Descriptions can be similarly obtained for various objects and concepts associated with a system. One may also view the actual source code using the hypertext interface, as shown in Figure 7. Declared symbols in the text appear as hypertext links; by traversing these links one can obtain information about the corresponding symbols.

5 Underlying Mechanisms

The following is a brief description of the representations and inference mechanisms used in the I-Doc software repository, and of the process of generating presentations.

5.1 Software repository mechanisms

Conceptually, the software repository is simply a collection of objects, each with a set of attributes and relationships with other objects. Some objects correspond to individual program components, and are represented

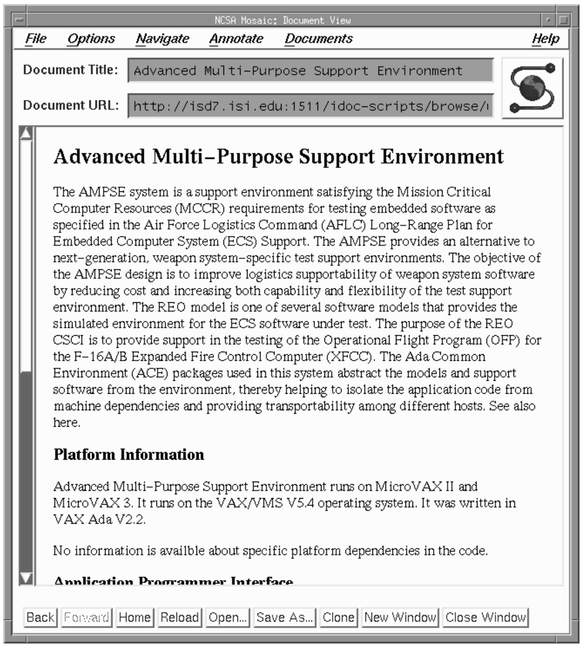

Figure 6: A non-task-oriented view

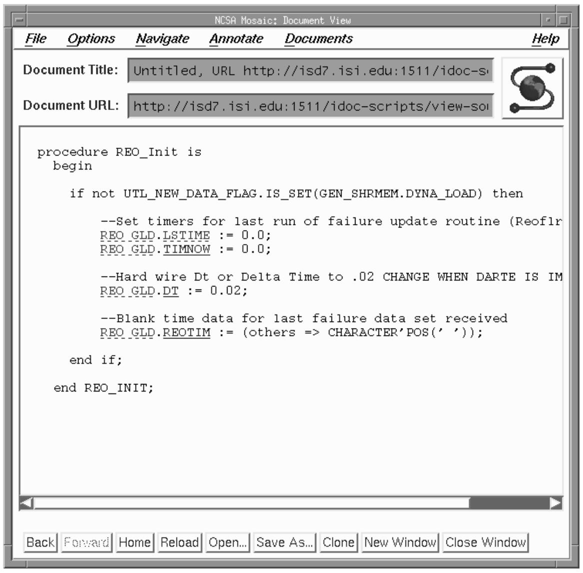

Figure 7: A hypertext view of program code

```
<NAME>REO Model External Routines</NAME>
<OVERVIEW>
These routines are called to make the AMPSE software simulate the
REO.  The executive calls the PROCESS_INPUTS to simulate inputing
data, then calls COMPUTE to simulate computations on the data, and
then calls PROCESS_OUTPUTS to simulate outputs in response to the
data.
</OVERVIEW>
<COMPONENT NAME="REO_Process_Inputs" PACKAGE="REO_EXPORT" TYPE="PROCEDURE-SPECIFICATION">
<COMPONENT NAME="REO_Compute" PACKAGE="REO_EXPORT" TYPE="PROCEDURE-SPECIFICATION">
<COMPONENT NAME="REO_Process_Outputs" PACKAGE="REO_EXPORT" TYPE="PROCEDURE-SPECIFICATION">
```

Figure 8: Internal representation of part of the AMPSE program interface

internally using Software Refinery's parse tree representation. Some objects contain information from multiple program components. For example, in Ada each package has a specification and a separate body. For presentation purposes, however, the package may be treated as a single object, whose properties are obtained from both the specification and the body. Other objects do not correspond to any program component at all, in which case a separate frame representation is used. The repository manages this heterogeneous representation so that the presentation generator need not be concerned with the internal representations employed.

Some object relations, such as component and definition-use relations, are derived from the source code. Other attributes are inserted as annotations. The annotation syntax consists of markup tags in the style of markup languages such as SGML. The tags are enclosed in angle brackets, and either represent the attribute values individually or serve as starting and ending delimiters surrounding the attribute value. This scheme is somewhat similar to the one used by the GRASP-ML system for annotating source code [3]. Attribute values may be text strings or collections of features used to identify the object that is the attribute value. Figure 8 shows the internal representation of the control portion of AMPSE's application programmer interface. This object corresponds to a collection of procedures in the source code which are not structured as a syntactic unit.

The repository also performs automated processing on object attributes. Some attributes are computed from more primitive attributes; some are defeasibly inherited from the attributes of other objects. For example, the attributes of an Ada procedure body may be derived from attributes of the corresponding procedure specification or from the package specification containing it. This capability is similar to the relation derivation mechanisms incorporated in the ARIES system [7], although somewhat limited in that no provision exists as of yet for editing derived attributes.

5.2 Presentation mechanisms

Presentations are generated by scripts written in the Perl language [16]. Perl was chosen because it is a high-level language somewhat comparable to Lisp, has strong string manipulation facilities, but does not produce large binary files. The scripts can executed on demand by the HTTPD server, without delays for system initialization as in Lisp.

Each script is supplied a set of parameters, some identifying the object to be described, and some describing the user making the request. These parameters are encoded in the World Wide Web addresses (URLs) for the hypertext links. Each script is responsible for generating URLs for follow-on questions, and associating them with hypertext links in the generated page.

Presentation generation occurs in the following phases. First, the script retrieves relevant attributes from the software repository. Next, the script determines what attributes need to be included in the presentation. Some of these attributes may need to be derived from primitive repository information, using natural language generation techniques. For example, a platform dependency "attribute" might be derived from lower-level information about what hardware platforms, compilers, etc. were used. Next, any embedded markup tags are converted into the HyperText Markup Language (HTML) used in the World Wide Web. Finally, the attributes are inserted into an HTML presentation template and transmitted to the user's World Wide Web client.

We are also experimenting with generating diagrams as part of our presentations. This involves generating Postscript commands for drawing primitives, appending them to a previously generated Postscript preamble, and transmitting to the Web client along with the textual information.

6 Conclusion

This paper has described efforts to analyze the inquiry process that is central to software understanding, and to build a tool which provides automated support for this inquiry process. Software understanding thus becomes less search-oriented, and more like a question-answer dialog.

The components of the system are currently undergoing trial evaluation. Groups outside of USC/ISI have expressed interest in using I-Doc for their own projects, and plans are in place for providing them with the system for their own use. Results from these evaluations should be available by the time this paper goes to press.

We are planning to extend the I-Doc work in several directions. First, we hope to enrich the software representation in the software repository, and enhance it with program transformations. Such transformations could automatically generate program slices relevant to a given inquiry, and extract more semantic information from the program code. Second, the natural language generator, which in the current system is fairly primitive, needs to be significantly expanded, and made flexible enough to select from among multiple presentation styles. Third, we wish to incorporate dynamically generated diagrams, as indicated above. Meanwhile further empirical studies of software inquiries will be conducted.

7 Acknowledgements

The authors wish to than Wright Laboratory for providing the software examples used in this paper. This work is sponsored by the Advanced Research Projects Agency and administered by Wright Laboratory, Air Force Materiel Command, under Contract No. F33615-94-1-1402.

References

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

[2] T.A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1990.

[3] J.H. Cross and T.D. Hendrix. Using generalized markup and SGML for reverse engineering graphical representations of software. In *Proceedings of the 2d Working Conference on Reengineering*, 1995.

[4] D.E. Egan. Individual differences in human-computer interaction. In M. Helander, editor, *Handbook of Human-Computer Interaction*, chapter 24, pages 543–568. Elsevier Science Publishers B.V. (North Holland), 1988.

[5] J.D. Herbsleb and E. Kuwana. Preserving knowledge in design projects: What designers need to know. In *INTERCHI'93*, 1993.

[6] W.C. Hill and J.R Miller. Justified advice: A semi-naturalistic study of advisory strategies. In *CHI'88*. ACM, 1988.

[7] W.L. Johnson, M.S. Feather, and D.R. Harris. Representation and presentation of requirements knowledge. *IEEE Trans. on Software Engineering*, 18(10):853–869, October 1992.

[8] A. Lakhotia. Understanding someone else's code: Analysis of experiences. *Journal of Systems Software*, 2:93–100, 1993.

[9] A.W. Lazonder and J. van der Meij. The minimal manual: is less really more? *Int. J Man-Machine Studies*, 39:729–752, 1993.

[10] D. Mayhew. *Principles & Guidelines in Software User Interface Design*. Prentice Hall, 1992.

[11] J.D. Moore. *Participating in Explanatory Dialogues*. MIT Press, Cambridge, MA, 1995.

[12] V. Rajlich, J. Doran, and R.T.S. Gudla. Layered explanation of software: A methodology for program comprehension. In *Proceedings of the Workshop on Program Comprehension*, 1994.

[13] P.L. Hunsaker R.E. Coffey, C.W. Cook. *Management and Organizational Behavior*. Austen Press, 1994.

[14] P.S. Selfridge. Integrating code knowledge with a software information system. In *Proceedings of the 5th Annual Knowledge-Based Software Assistant Conference*, pages 183–195, Syracuse, NY, 1990.

[15] E. Soloway, J. Pinto, S.I. Letovsky, D. Littman, and R. Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11), November 1988.

[16] L. Wall and R.L. Schwartz. *Programming perl*. O'Reilly & Associates, Sebastopol, CA, 1991.

[17] P. Wright. Issues of content and presentation in document design. In M. Helander, editor, *Handbook of Human-Computer Interaction*, chapter 28, pages 629–652. Elsevier Science Publishers B.V. (North Holland), 1988.