Task Oriented Software Understanding

Ali Erdem, W. Lewis Johnson, Stacy Marsella
USC Information Sciences Institute & Computer Science Dept.
4676 Admiralty Way
Marina del Rey, CA 90292-6695
+1 310 822 1511
{erdem,johnson,marsella}@isi.edu

ABSTRACT

The main factors that affect software understanding are the complexity of the problem solved by the program, the program text, the user's mental ability and experience and the task being performed. This paper describes a planning approach solution to the software understanding problem that focuses on the user's task and expertise. First, user questions about software artifacts have been studied and the most commonly asked questions are identified. These questions are organized into a question model and procedures for answering them are developed. Then, the patterns in user questions while performing certain tasks have been studied and these patterns are used to build generic task models. The explanation system uses these task models in several ways. The task model, along with a user model, is used to generate explanations tailored to the user's task and expertise. In addition, the task model allows the system to provide explicit task support in its interface.

Keywords

software explanation, software understanding, user model, task model, knowledge representation

INTRODUCTION AND MOTIVATION

Software maintenance has become an important activity in the software industry. Maintenance of existing systems consumes 50% to 75% of the total programming effort [16] and a significant portion of this maintenance activity (30% to 60%) is spent on software understanding [5, 6].

Software Understanding is the reconstruction of logic, structure and goals that were used in writing a program in order to understand what the program does and how it does it [2, 3]. This reconstruction process is typically composed of inquiry episodes [3, 15] which involve the following steps: read some code, ask a question about the code, form an hypothesis and search the documentation and the code to confirm the hypothesis. The generation and verification of the hypothesis are influenced by the salient features in the code and the documentation [3].

Attempts to solve the software understanding problem has focused on two methods: improving the search and automating the recognition of features in the code. The search process was improved either by providing an automated search capability or by changing the organization of the documents. For example, Devanbu's LaSSIE system [6] used description logic to represent the domain and the basic software knowledge. Users could do searches by forming queries using the predefined domain and software concepts.

Linked and layered documentation organizations have also been used to improve the search process. Soloway linked parts of the documentation to delocalize the programming plans [15]. SODOS [19] project at USC linked all the Software Life Cycle documents of a software project and also provided search capabilities. Rajlich [12] organized documentation into a problem domain layer, an algorithm layer and a representation layer. Users easily guessed which layer would have the answer to their queries and restricted their search to that layer.

The Programmer's Apprentice [13], Hartman's UNPROG [8] and Will's GRASPR [20] programs tried to automate the recognition of the standard programming plans in the code. This approach improved both recognition of the features in the code and reduced the search space from the actual code to the programming plans.

These improved search and automated recognition methods ignored one important factor, the user. Software understanding is affected by the complexity of the problem solved by the program, the program text , the user's mental ability and experience, and the task being performed [3]. The methods described above focused on the first two factors, but ignored the latter two. Even with good documentation, users still prefer asking questions to system experts or other users before consulting the documentation. There are several reasons for this behavior. First, the dialogue between the user and the expert facilitates the refinement of the questions [21]. The interactive nature of the dialogue also permits the users to ask follow-on questions and clarify the parts they did not understand. In addition, the human experts can recognize the users' plans and provide answers to satisfy their goals [4]. Finally, The experts also rec-

ognize the users' level of expertise and provide tailored answers that are easier to understand.

Our research goal is to develop an interactive software explanation tool that can act as a system expert and provide tailored explanations to user questions that are easy to understand and are relevant to the user goals. Such a tool needs to have the necessary knowledge to answer the user questions. To identify what type of questions are asked by the users and what governs the answers the experts provide, we studied the questions posted to the USENET Tcl/Tk newsgroup [9]. Based on this study and a survey of the research literature on questions, we later developed a domain independent question model for software understanding questions.

The results of the USENET study are being used in the implementation of an online documentation tool called MediaDoc. MediaDoc's knowledge representation is organized around the question model. Further, a planning component has been added to MediaDoc for user tailoring. The planner uses the question model to determine the explanation content. Finally, to address the knowledge acquisition problem we are exploring the utility of using the question model as an upper model in a text extraction subsystem.

The USENET study also revealed the importance of users' plans and goals in both the questions and the answers. The most frequently asked questions included descriptions of what the users' goals were. If the users had attempted to solve the problems themselves, the messages also included their plans. In addition, the experts' tailored their answers by including only the information relevant to the user's goals and plans.

Users in the study described their goals in detail because there was no surrounding task context. This placed an overhead on the interaction. Clearly, an automated explanation tool which required the users to specify their goals in detail at every step would be unusable. To address this problem we needed a better understanding of the dynamics of the inquiry episodes and their relationship to the user's task. However, the USENET messages typically contained only a single inquiry episode and were not very useful for such a study.

To determine the dynamics of the inquiry episodes and to investigate the types of questions users ask within the context of a task, we decided to study users while they performed a task. We were particularly interested in finding out how the users started their task, the goals they tried to achieve at each step and the relation between the user goals and the questions they asked.

We studied two users while they performed different tasks on two different large software systems and recorded the questions they asked. Although the program domains were different, there was significant overlap in terms of the types of questions asked and the goals of the task steps. We are using these patterns in user questions and goals to build a task taxonomy. This task taxonomy can be exploited in several ways. First, we plan to use this taxonomy in MediaDoc to determine the user goals and tailor the explanation accordingly. Second, this taxonomy is used for building generic task models, which MediaDoc uses to give explicit task support when needed.

In this paper, we will describe the question model and user tailoring briefly to demonstrate how MediaDoc works. The focus of the paper however will be on the task study. We will present the details of the study and the observations we made. We will then describe how these observations will affect MediaDoc.

QUESTION MODEL

Questions are the basis of user's interaction with the documentation and the system experts. Wright claimed that people's interaction with documentation starts with formulating a question, therefore the documentation content needs to be determined by the questions users ask [21].

We studied the questions asked by Tcl/Tk programmers in a USENET newsgroup and identified the most commonly asked question types [9]. It was possible to request the same information in many different ways in natural language. For our purposes, *What does X do?*, *What is the function of X?*, *What does X cause?* all requested the same type of information. We classified the questions in the data set based on the type of information requested. In addition, since the USENET data set was biased and did not include questions for all software engineering tasks [9], we decided to survey the research literature for other studies on questions. After reviewing Lehnert's [10] and Graesser's [7] question answering systems, Swartout's research on questions asked during expert system explanations [17], Letovsky's research on questions asked during inquiry episodes [11] and Serbanati's list of most commonly asked types of information by programmers [14], we developed a question model. In this model, a question is represented based on its topic, question type and the relation type.

- Topic: The question topic is the entity referenced in the question. It can easily be identified as the subject of the question. For example, in *What does procedure open do?*, the topic is *procedure open*.
- Question type: The question type identifies the type of information requested. It is one of verification (is), identification (what), procedural (how), motivation (why), time (when) or space (where).
- Relation type: The relation type identifies what

Table 1: Simple Questions

	self	input	output	structure	cause	use	goal	require	satisfy	context
Verification (Is)	Does it exist?	Does it have inputs?	Does it have outputs?	Does it have structure?	Does it do anything?	Is it used?	Does it have a goal?	Does it have requirements?	Does it satisfy anything?	Does it have a context?
Identification (What)	What is it?	What are its inputs?	What are its outputs?	What is its structure?	What does it do?	What uses it?	What is its goal?	What does it require?	What does it satisfy?	What is its context?
Procedural (How)	How does it work?	How are the inputs processed?	How are the outputs processed?	How is it structured?	How does it do?	How is it used?	How do the goal arise?	How can the requirements be met?	How are the conditions satisfied?	How does it interact with its context?
Motivation (Why)	Why is it necessary?	Why are the inputs necessary?	Why are the outputs necessary?	Why is it structured?	Why does it do?	Why is it used?	Why is the goal necessary?	Why are the requirements necessary?	Why are the satisfied conditions necessary?	Why is the context necessary?
Time (When)	When does it exist?	When are the inputs provided?	When is the output received?	When is it structured?	When does it work?	When is it used?	When does the goal arise?	When do the requirements arise?	When does the post conditions arise?	When is the context ready?
Space (Where)	Where does it exist?	Where are the inputs?	Where are the outputs?	Where is it structured?	Where does it work?	Where is it used?	Where does the goal arise?	Where are the requirements?	Where are the post conditions?	Where is the context?

kind of information about the question topic is requested. We identified the most commonly asked relation types from our data set, but further additions to this set are possible. The identified relation types are as follows:

- *Topic (self):* These questions ask about the question topic, e.g. *What is an integer?*, *How does function X work?*
- *Behavior (input/output):* These questions ask about the input/output relationships, e.g. *What are the inputs to X?*
- *Structure:* These questions ask about the structure of the topic, e.g. *What are the components of Y?*
- *Function (cause):* This type of questions ask about the causal relations between topics, e.g. *What does X do?*
- *Use:* This type of questions ask about the usage of the topic, e.g. *What uses X?*
- *Goal/Purpose:* These questions ask about the goal of the topic, e.g. *What is the goal of X?* The goal relation type and the motivation question type are different in the sense that one asks for the purpose of the topic whereas the other asks for the reason of existence.
- *Require:* These questions ask about the requirements (preconditions) of the topic, e.g. *What needs to be done before function X is called?*
- *Satisfy:* These questions ask the post conditions the topic satisfies, e.g. *What is true after function X is called?*
- *Context:* These questions ask about the relationship between the topic and its environment, e.g. *Does X windows require a specific operating system?*

More question types and relation types can be added to this model if necessary. For example, a *Who* question type can be added to answer questions about the ownership of topics. Some question type, relation type pairs do not apply to all topics, e.g. data items do not have processes and it is not very meaningful to ask how an integer variable works.

We categorized the questions into two groups based on how the answers can be calculated:

- *Simple questions:* These questions can be answered by simple data retrieval and are shown in table 1. Simple questions are represented as a three tuple (topic, question type, relation type), e.g. *How does procedure open work?* is represented as (procedure open, how, self), *What are the parts of student record?* is represented as (student record, what, structure).

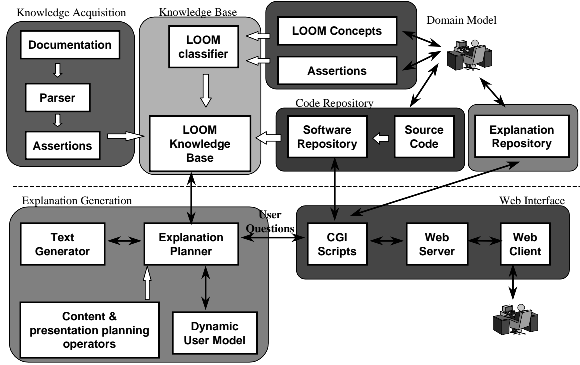

Figure 1: MediaDoc Architecture

- *Complex Questions:* These questions require a data retrieval and evaluation of the retrieved data to answer the question. Complex questions are represented as a four tuple, (topic, question type, relation type, evaluation function) where the evaluation function is a user supplied, predefined function. Some of these predefined functions are as follows:
  - *Count: How many inputs are there to function X?* is represented by (function X, what, input, count).
  - *Significance: What is the most important component of X?* is represented by (X, what, structure, importance).
  - *Comparison: What is the difference between X and Y?* is represented by (X and Y, what, self, difference).

USER TAILORING

The current MediaDoc architecture is given in figure 1. User tailoring is done by the explanation generation component which will be the focus of this section. When a user asks a question, this question is communicated by the web interface to the planner. The planner then creates a top level goal for explaining the question to the user. This top level goal is represented as (knows user topic question-type relation-type). For example, when the user guest asks *How does function-X work?*, the top level goal will be (knows guest function-X how self).

Then, the planner tries to create an explanation plan using the plan operators. The plan operators interact with the user model and the domain model to create a user tailored explanation plan. The user model contains information about the user's domain knowledge, his preferences, his task and level of experience. For example, if user guest is a novice programmer and knows what function-X is the user model will contain the facts

```
:operator inform-how-it-works
:parameters (?user ?topic ?experience ?task)
:precondition
 (and (verbosity ?user high)
      (knows ?user ?topic what self))
:effect (knows ?user ?topic how self))
```

Figure 2: A sample plan operator

(expertise guest novice), (task guest programmer) and (knows guest function-X what self).

A sample plan operator is shown in figure 2. This plan operator can be summarized as *if a user asks how something works and if he prefers detailed explanations then explain what that thing is first if he doesn't already know.* So in our example above, if the user guest did not know what function-X was then the system would have explained what the function-X is before explaining how it works. After an explanation, the user model is updated to reflect the fact that user guest now knows how function-X works. Note that, since the user guest knew what function-X was, the precondition of this plan operator would have been satisfied and as a result the system would not have explained what the user already knew.

Some of the tailoring methods implemented by the current plan operators are as follows:

- Don't tell the user what he already knows
- Tailor the explanation to the user's role, e.g. if an end user asks what something is, describe it in terms of what it does.
- Tailor to the user's expertise, e.g. if a novice asks about a topic, describe the topic in general terms and not in system specific terms.
- Tailor to the domain knowledge, e.g. if a user knows how all the components of a topic works and how the topic is structured, he also knows how the topic works.

TASK MODEL

Users frequently described their plans and what they were trying to achieve in the messages posted to USENET Tcl/Tk newsgroup. The answers provided by the experts were also tailored to the users' goals. This showed the importance of producing explanations that are tailored to the users' goals and plans. However, it is unreasonable to expect the users to specify their goals at every step while using an online explanation system. Users do not need to specify their goals at every step when they consult a system expert, because

both the system experts and the users share common human problem solving knowledge.

Identifying the user task is one way of finding out the user goals. We define user task as the operations the user wants to perform on a set of topics, e.g. *modify route behavior, write an interface function, integrate radar and display components etc.*. Each task have an associated user goal. Some other observations about user tasks are as follows:

- *Task determines the question topics:* Not all topics are relevant to the user task. Once the user decides to perform a particular task, he will focus on a set of topics and ask questions about them.
- *Task affects the types of questions asked by users:* Similarly task affects the types of questions the users ask. For example, a system integrator is interested in the system architecture and how the components fit together, so he will ask questions about the structure of the system and the input output behavior of individual components. A programmer working on an individual component on the other hand is going to ask questions about the functions of that component and how the component works.
- *Novices require help for performing the task:* Novices ask questions like *What do I do now?* that can only be answered if the system has knowledge about the task. Experts also benefit from this task information when it's made explicit during the task, since it reduces their short term memory load.

The users go through many inquiry episodes while they perform a task. We were interested in finding out the dynamics of these episodes in order to learn how to guide the user to his goal and also to prevent users from taking explanation paths which fail to address their goals. The USENET data did not help us much for this, because each message in that study contained information only about a single inquiry episode. Every question in our model is composed of two major components: the topic and the question part. When the user moves from one inquiry episode to another either the topic or the question or both the topic and the question part can change. To understand the dynamics of this behavior better, we decided to study the user questions while they performed a certain task. We were particularly interested in finding the answers to the following questions:

- Where do the users start their task? How do they identify the system components relevant to the task at hand?
- Is there a structure to the task being performed? Do the users move from one inquiry episode to another based on some criteria? Do they try to satisfy particular goals at each step?
- What type of goals do the users try to achieve at each step? What type of questions do they ask? Is there a relation between the type of the questions and the goals?
- Can all user questions be represented by our question model?

To find the answers to these questions, we studied two users performing different tasks on two large software systems. In the first study, the task was to make a modification to a large software system and analyze the impact to other system components. The user performed the task in a day and recorded the individual steps he took along with the reasons for taking them. In the second study, the user was given a report of a problem that happened during system integration and was assigned the task of finding the source of the problem. In this study, the user did not perform the task, but rather described the interviewer the steps he would take and the questions he would ask at each step.

ModSAF Impact Analysis Study

ModSAF (Modular Semi-Autonomous Forces) is part of a distributed simulation (DIS) system for training that creates a virtual environment where trainees in simulators can interact with virtual automated forces which ModSAF simulates. ModSAF is written mainly in C and the version we were employing (version 2.1) had over 250 libraries and over 1.5 million lines of code.

The task to perform was to study the impact of creating an interface to the aircraft simulation components that would allow the dynamic modification to the route flying behavior. The subject's knowledge of ModSAF was based on five months experience in ModSAF, modifying it to build an interface that reported back the status of automated ground vehicles. Included in that knowledge was the awareness that ModSAF's simulations of an automated entity performing some mission were built from collections of lower level behaviors called tasks. The resources available for this impact analysis included the source code and the documentation which was in GNU info file format. In addition, the subject had available, and knew how to use, a specially instrumented version of ModSAF that could report what low level tasks were being invoked when a vehicle was performing some mission.

The subject took the following steps to complete the task:

- **Step 1. Comprehend the desired behavior:**

The first step the subject took was to understand the desired modification. He was already familiar with ModSAF and knew what was meant by the dynamic route flying behavior.

The questions he asked in this step were *What is meant by dynamic modification to the route flying behavior? What is the desired behavior of the modified system?*

- **Step 2. Identify the relevant components:** The second step the subject took was to identify the relevant system components. The subject ran a special instrumented version of ModSAF, which told him what ModSAF tasks had started or stopped, in order to identify the libraries that were involved in route flying behavior. The subject ran two different missions, a sweep mission and an attack ground target mission to identify the libraries. Running two missions was necessary for identifying if there was more than one implementation of vehicle navigation in ModSAF.

  The question he asked in this step was *What libraries are used in implementing route flying behavior?*

- **Step 3. Test whether route modification will work:** In this step, the subject modified the routes using ModSAF's task modification and GUI interface while the missions were running. His goals were to identify whether it was possible to modify the routes at all and also to find out the libraries involved in the route modification. Route modification with task modification interface was successful, but changing the route through the GUI interface failed.

  The question he asked in this step was *Is it possible to change the flight routes during a mission?*

- **Step 4. Comprehend unknown concepts:** The subject found out the names of the libraries involved in the previous steps and using ModSAF's naming conventions, he was able to find the documentation for these libraries. He read the documentation to understand what these libraries were and how they worked.

  The questions he asked in this step were *What is library X? What does it do? How does it work?*

- **Step 5. Comprehend how the system currently works:** In this step, the subject looked at the implementations of the libraries he thought were relevant to the route flying behavior. He looked at the route data structure, the parameters of the flight task data structure and the finite state machine code for the route flying task. He investigated why route modification through the task modification interface was successful and why GUI interface modification failed. He concluded that routes were assigned persistent object ids and in GUI modification case this id becomes invalid resulting in termination of route flying. He also investigated how groups of vehicles follow a route.

  The questions he asked in this step were *How is route implemented? What is the data structure for a route? What are the parameters to the task data structures? What are the dependencies between libraries?*

- **Step 6. Find methods for implementing the desired behavior:** Based on his observations, the subject came up with a list of suggestions for implementing the desired behavior. He did not actually implement the suggested modifications. He used his general programming knowledge to come up with the suggestions. He did not record any questions in this step, but we would expect him to ask some questions, especially when there are alternatives for implementation.

Northrop-Grumman Integration Task Study

The system used in the second study was B2 aviation software. The engineer who performed the task was an expert in this area. He was given the problem report of a post integration problem on the airplane and was asked to find the source of the problem. The problem was a corrupted display service message during initialization. It only occurred once during the initialization stage and was a transient problem. The effect was either the loss of radar video or connection of the display unit to the wrong radar buffer.

The information sources that were accessible by the engineer were the video recording of the problem, the message bus traffic log from the airplane, a message database which contained the descriptions of all message types, the requirements documentation, the program code and the system experts, i.e. programmers and requirements people involved. The subject was not familiar with the details of the radar and display unit components, but he was familiar with the overall system architecture and had extensive knowledge in troubleshooting these kinds of problems.

The subject did not actually perform the task, but described the interviewer the steps he would take and the questions he would ask at each step.

- **Step 1. Identify the problem:** The first step was to understand exactly what the problem was. The engineer read the problem description carefully and identified all the terms he did not know. He would talk to the experts who wrote the problem

description and the programmers to clarify the unknown or ambiguous terms.

Some of the questions he asked in this step were *What is term? What does term do? What is the relationship between the term and the problem?*

- **Step 2. Comprehend the desired behavior:** In the second step, the subject wanted to know the desired behavior. He would consult with the experts and the requirements documentation to find the answer. He would first attempt to do a keyword search in the requirements documentation. If the keyword search fails to produce an answer, he would do a sequential search on the table of contents. His search would also attempt to identify the messages related to the correct behavior from the requirements documents. The search process was an iterative one. After reading a requirements document found with a keyword search, the subject would add new keywords to his list. The process would continue until all the relevant requirements were satisfactorily covered.

  The questions asked in this step were of the form *What are the requirements that include these keywords? Is the requirement relevant to the problem? What does the requirement say about the desired behavior?*

- **Step 3. Identify the messages used in implementing the requirements:** At this point, the subject had a good understanding of the desired behavior. He assumed the problem was due to incorrect implementation of requirements, since if it were due to incorrect requirements he would have noticed it in the previous step. So he started the search to identify the messages used in implementing the behavior. His main information source in this step was the message database. He would do a keyword driven search in the message database to find out which messages were involved with the behavior.

  The questions he asked in this step were in the form *What message descriptions include this keyword? Is this message relevant to the problem?*

- **Step 4. Test the hypotheses generated in previous steps:** In this step, the subject tested the hypotheses generated in previous steps and asked whether he found the source of the problem. The steps were not necessarily executed sequentially, that is the subject could investigate a couple of messages, form an hypothesis and execute step four to verify whether he had found the source of the problem.

  The main question he asked in this step was *Have I found the source of the problem?*

- **Step 5. Identify the problems in the bus traffic:** If the problem had not been solved yet, the subject would generate new hypotheses by looking at the bus traffic. He had some heuristics to identify the problems. For example, he would check to see the correct ordering of the messages, he would look for unexpected separation of messages or he would check if the problem was caused by a timing dependent message ordering. He formed this problem solving knowledge in previous troubleshooting episodes.

  The questions he asked in this step were mainly verification questions in the form of *Does this happen in the bus traffic?*

- **Step 6. Identify similar problems:** If everything else fails, the subject would ask the developers and other experts if they had seen anything similar to this problem. This was another method for hypothesis generation. He would also attempt to run an instrumented version of the avionics software in the lab to reproduce the problem and capture more data about the problem. This would also enable him to generate new hypotheses.

  The main question in this step was *What could be wrong?*

Observations

- *Were there similarities between the two tasks?* Although the tasks and the problems were different, there were similar steps in the problem solving behavior. These were not necessarily executed in the same order, but both subjects had a *gather* step in which they identified the relevant concepts, a *comprehend* step in which they read the documentation to understand these concepts, a *test* step in which they tested the hypotheses. The second subject was required only to find the source of the problem. His task did not include making modifications. That's why he did not have a *find method* step like the first subject.

- *Where do users start their task?* Both subjects had an initial information gathering step to identify the relevant topics. The first subject ran an instrumented version of ModSAF to identify which tasks started and stopped. The second subject preferred a keyword search on requirements to start. There are many ways of satisfying information gathering goal and some of these are domain dependent. For example, an instrumented version of the program might not be available to all the users. However, some techniques like keyword searches are domain independent and would be useful in all domains.

The method used for information gathering depends both on personal preferences and the costs associated. For example, in Northrop study running the instrumented avionics software in the lab was costlier than doing keyword searches on the requirements and was not preferred.

- *Was there a structure to the task being performed?* There were similar steps used by both subjects and the problem solving behavior seemed to be structured. The ordering of the steps, however, was not exactly the same and the subjects did not execute the subtasks sequentially.
- *Was there a relation between the goal and the type of question?* The subjects asked similar questions during the subtasks. *Gather* task was addressed by the search process, *comprehend* asked *What is (term)?* and *How does (term) work?* types of questions and *test* asked verification questions.
- *Sufficiency of the question model:* Most of the questions asked by the subjects were supported by the question model, but there were also some problems. These problems were as follows:
  - Weak support for *gather* task: The question model requires a well defined question topic and does not support *gather* type of questions. These questions are intended mainly for search and usually include a description or some other search criteria. However, there are search mechanisms which do not use the question model in MediaDoc, and these types of questions are supported by them. Another option of supporting these questions is to extend the question model and allow descriptions and search criteria to be given as a topic. A topic resolution component can then do the search for the user and find the corresponding topics that satisfy the search criteria.
  - Complex topic descriptions are problematic: The question model shifts the complexity of the topics to the domain model. So the question *Is it possible to change the flight routes during a mission?* can be modeled as a complex question, e.g. (changing the flight routes during a mission, is, self, possible). However, extensions to the domain model are necessary for representing such complex topics.
  - No support for *Have I found the source of the problem?* questions: Some of the questions require external evaluation and decision making capabilities. These questions are not supported well by the question model. In general, the *test* task step requires such external evaluations and are not well supported.
  - No support for *What could be wrong?* questions: These questions require the system to have problem diagnostics skills and are hard to answer without such a component. Our goal in MediaDoc is not to build a diagnostics tool, but if we had such a tool it could have been used to answer these types of questions.
- *Hierarchical and multi-layered mental representations:* The programming process constructs mappings from the problem domain to the programming domain, possibly through several intermediate domains and software understanding is the reconstruction of these mappings [3]. As a result, the experts' mental representation of computer programs are hierarchical and multi-layered [18]. In our studies, we observed confirming evidence for this hypothesis. The first subject constructed mappings from the vehicle behaviors of the running program to ModSAF tasks, from ModSAF tasks to finite state machines and finally from the finite state machines to the code. The second subject constructed mappings from the requirements to messages, and from these messages to the code. Both subjects' hypothesis generation was top down and driven by these layers.
- *Heuristics for ordering hypotheses:* Both subjects used some criteria for ordering and testing hypothesis. The first subject looked first at the route behavior of individual units then groups. His ordering criteria was based on the complexity of the behavior. The second subject explained the criteria he used as "How likely it is for this to be the problem and how easy it is to test it". He assumed that the problem wouldn't be trivial, since the programmers usually did not make trivial errors and the components were already unit tested before the integration.
- *Users desire automated support for mundane tasks:* Both subjects claimed that their performance would have improved if they had a tool to support their *gather, identify* tasks. They both had to shuffle through the documentation and do keyword searches to find out the information they were searching.

Software Engineering Tasks

We are using the results of the task study for building a task taxonomy. In this taxonomy, each task will be associated with a specific user goal. Since, studying two tasks is not sufficient for identifying all possible task steps and the user goals addressed by these, we investigated the software engineering task descriptions used by other researchers. The taxonomy in figure 3 is prepared after reviewing the task definitions in Serbanati

```
Analysis
   identification
   comprehension
   classification
   assessment
Synthesis
   design
   planning
   modeling
Fault finding
   diagnosis
   repair
   verification
Configuration
Modification
Informative
   Specification
   Documentation
   Explanation
Management
```

Figure 3: A taxonomy of software engineering task

[14], CommonKADS [1] and Programmer's Apprentice [13]. In this taxonomy, *gather* is an identification task, *comprehend* is a comprehension task and *test* is an assessment task.

IMPLICATIONS FOR MediaDoc

We are incorporating the results of the question and the task studies into MediaDoc. Some of the implications these studies have on MediaDoc are as follows:

- *Explicit support for the user task:* Explicit task support is critical for novices and useful for experts. We are incorporating task support in MediaDoc using domain independent generic task models. These models are based on the task taxonomy. When a user wants to perform a task on a topic, the task model will be instantiated and placed in the task agenda. This way, both the user will know the task steps and MediaDoc will be able to identify the user goals for each task step.

- *Support inquiry episodes using the question model:* The users generate and test many hypotheses during software understanding. These inquiry episodes are going to be supported in MediaDoc in two different ways. First the users will be able to ask questions about any topic using the question model. Second, MediaDoc will keep track of the future user questions in the task agenda. The user can then focus on which question to investigate next rather than trying to remember all the questions he have.

- *Tailor the explanation to the user goals:* The task steps in the generic task model will have distinct goals. MediaDoc will tailor the explanation to the users' goals at each task step. For example, in the *gather* task step, MediaDoc will emphasize the relevance of the topics, in *comprehend* task step it will explain what the topic is and how it works.

- *Support for gather task:* Users start their task by an information gathering step. Support for this step needs to be built in MediaDoc. Techniques like keywords searches and query mechanisms are being enhanced to support the *gather* task.

CONCLUSION

Software understanding has become an important problem in the software industry. Better tools are needed for solving this problem and these tools can be developed only after we study how the users understand software. Our research goal is to build a tool that supports human software understanding. We studied user questions within and outside the task context and found out that interactivity, user and task tailoring are important requirements for such a tool. We took a planning approach solution to address these requirements and were successful in producing explanations tailored to the user.

We will work on incorporating task oriented explanations to the tool we developed. Then we will evaluate the effectiveness of our tool in software understanding.

ACKNOWLEDGMENTS

This work is sponsored by DARPA under DARPA order number D880.

REFERENCES

[1] Agnar Aamodt, Bart Benus, Cuno Duursma, Christine Tomlinson, Ronald Schrooten, and Walter Van de Velde. Task features and their use in CommonKADS. Technical report, Vrije Universitiet Brussel, University of Amsterdam, 1993.

[2] Deborah A. Boehm-Davis. Software comprehension. In M. Helander, editor, *Handbook of Human-Computer Interaction*, chapter 5, pages 107–121. Elsevier Science Publishers B.V. (North Holland), 1988.

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

[4] Sandra Carberry. *Plan Recognition in Natural Language Dialogue.* MIT Press, 1990.

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

[6] Premkumar T. Devanbu. *Software Information Systems*. PhD thesis, Rutgers, The State University of New Jersey, New Brunswick, October 1994.

[7] Arthur C. Graesser, Paul J. Byrne, and Michael L. Behrens. Answering questions about information in databases. In Thomas W. Lauer, Eileen Peacock, and Arthur C. Graesser, editors, *Questions and Information Systems*, chapter 12, pages 229–252. Lawrence Erlbaum Associates, Publishers, 1992.

[8] John Hartman. *Automatic Control Understanding for Natural Programs*. PhD thesis, The University of Texas at Austin, May 1991.

[9] W. Lewis Johnson and Ali Erdem. Interactive explanation of software systems. In *KBSE '95*. IEEE, 1995.

[10] Wendy G. Lehnert. *The Process of Question Answering: A Computer Simulation of Cognition*. Lawrence Erlbaum Associates, Publishers, 1978.

[11] Stanley Letovsky. Cognitive processes in program comprehension. *The Journal of Systems and Software*, (7):325–339, July 1987.

[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] Charles Rich and Richard C. Waters. *The Programmer's Apprentice*. ACM Press, 1990.

[14] L.D. Serbanati. *Integrating Tools for Software Development*. Yourdon Press, 1993.

[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] I. Sommerville. *Software Engineering*. Addison-Wesley, 4th edition, 1992.

[17] William R. Swartout and Johanna D. Moore. Explanation in second generation expert systems. In Jean Marc David, Jean-Paul Krivine, and Reid Simmons, editors, *Second Generation Expert Systems*, chapter 24, pages 543–585. Springer-Verlag, 1993.

[18] Susan Wiedenbeck and Vikki Fix. Characteristics of the mental representations of novice and expert programmers: an empirical study. *International Journal of Man-Machine Studies*, 39:793–812, 1993.

[19] Ronald Charles Williamson. *SODOS - A Software Documentation Support Environment*. PhD thesis, University of Southern California, Los Angeles, December 1984.

[20] Linda Mary Wills. *Automated Program Recognition by Graph Parsing*. PhD thesis, Massachusetts Institute of Technology, July 1992.

[21] 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.