A Little Knowledge Can Go a Long Way Towards Program Understanding[1]

Jelber Sayyad-Shirabad
Timothy C. Lethbridge

Department of Computer Science
University of Ottawa
Ottawa, Canada, K1N 6N5
{jsayyad, tcl}@csi.uottawa.ca

Steve Lyon

Mitel Corporation
Kanata, Ontario, Canada
Steve_Lyon@software.mitel.com

Abstract

Large, complex software systems are hard to learn and navigate. In an ideal environment, documentation can help in this process. However the latter is usually out of date and hard to use. Others have proposed using large knowledge bases to model software systems, however these are very expensive to build and may be as unmaintainable as the code. In this paper, we propose instead to use a highly circumscribed, small, conceptual knowledge base, whose purpose is to help the apprentice navigate a software system, and facilitate search within the code. We present our vision, and some initial experiments which involve building such a knowledge base in a semi-automated way.

1. Introduction

This paper presents an approach to program understanding that makes use of a very simple knowledge base.

We hypothesize that maintainers will more effectively understand source code if the browsing, search and analysis tools they use were able to make use of a knowledge base. Such a knowledge base would describe only the high level system concepts and properties and would be kept very small so it could be maintained.

In this paper, we present this approach in some detail and describe an application of the approach in an industrial company. We place emphasis on the knowledge acquisition approach we have used – a semi-automated method designed to minimize consumption of the maintainers' scarce time.

2. Overview of the approach

2.1 The scenario

In industrial projects which we have studied, the following scenario is very common:

- A group of software engineers (SEs) are evolving a large legacy system, that is hard to maintain due to its complexity and continuous change.
- Documentation exists, but is little used for program understanding because of one or more of the following: It is a) out of date, b) hard to maintain, c) difficult to understand, d) difficult to access, and/or e) not structured so as to be useful to typical maintenance problems.
- The information sources maintainers use most are: a) the source code, b) their mental model of the system, as developed over time by experience , and c) the knowledge of other SEs.
- The mental models of 'apprentices' (who may be expert software engineers, but who are new to the particular system) are naturally non-existent or poor. The mental models of experts tend to focus on particular parts of the system that they have been working on more frequently.

The initial subject of our studies is a maintenance team within Mitel corporation, a telecommunications company. They are working on a large real-time system that has been evolving since the early 1980's and still produces a significant component of company revenue. The above four points are clearly true in this team.

The goals of our work are twofold: a) To improve the productivity of expert maintainers, and b) To reduce the amount of time it takes an apprentice to learn the system.

The work reported in this paper is part of a larger project [1] that has two main thrusts: 1) We study software engineers scientifically, to learn their work

[1] This work is supported by NSERC and Mitel Corporation and sponsored by the Consortium for Software Engineering Research (CSER).

patterns and needs; 2) We apply the results of this study to develop tools that satisfy the above goals. Our overall approach to tool creation uses user-centred design techniques.

2.2 Some solutions proposed in the past

The following are some alternatives to our approach, with arguments as to why they are insufficient:

2.2.1 Use a more sophisticated knowledge base. We believe that once a body of documentation grows beyond a certain size, its 'marginal usefulness' diminishes to the point where the cost of production and maintenance drops below the benefit. A knowledge base is really just a structured kind of documentation that the computer can process: We therefore believe it is futile to build a descriptive knowledge base that is very large [2, 3]

We also observe that software engineers tend to make the vast majority of their references to a small subset of the documentation, e.g. conceptual overviews or maintenance handbooks [4]. This appears to be because what the SEs really need from documentation are 'key' items of information, not a large amount of detail. They can often obtain the detail more easily by examining the source code.

2.2.2 Improve the conventional documentation. It might be argued that one should simply improve existing documentation, and perhaps add hyperlinks from documentation into code. We agree that this is a useful approach, however we propose that the presence of formal relationships in a knowledge base is of more value than mere descriptive text or non-computer-interpretable diagrams. We have demonstrated this in earlier research [5, 6].

2.3 Generic description of the knowledge bases

Knowledge bases used in our approach have the following characteristics:

1. They are based on concepts and conceptual relations. They do not contain rules, procedures or complex logical conditions. For our current work, we are using a knowledge management system called CODE4 [7, 8].
2. They are small (200-400 main concepts).
3. They cover the most important high-level concepts of the system under maintenance.

A new knowledge base will have to be created for each new system to be understood. Each existing knowledge base evolves as its corresponding system changes – some of this maintenance can be automated.

We have used such knowledge bases to help in the design of software [5] We are now adapting the same technology to its maintenance.

2.4 Application of the knowledge bases to program understanding

Once a knowledge base is constructed, we plan to develop tools that make use of it. The following sections describe situations where we hypothesize such knowledge bases could be employed. We are planning to validate these hypotheses using experiments and observation. Also note that these tools will fit within the context of a larger architecture which is shown in figure 1.

2.4.1 Improved navigation. 'Apprentices' will be able to browse the CODE4 knowledge base in a systematic way. When they encounter a concept, they will be able to see its description, how it is implemented, pointers to relevant documentation and other related concepts. We hypothesize that they will be able to learn about code faster by alternating between the implementation and conceptual levels. All classes of SEs will be able to use the knowledge base to help work with more abstract views of the system.

2.4.2 Improved search. Searches within code often fail because the software engineer does not know the precise way to which a concept is referred in the code.

If a tool can access a knowledge base, it can intelligently suggest other terms that might be used as alternate search keys. For example, imagine somebody searches for 'call waiting' because they are used to a telephony feature commonly available on central offices. In the context of a specific PBX system, this concept may not be found in the source code. However, the knowledge base might 'know' that a related concept is 'camp-on', and point the software engineer to that instead. Another closely related scenario is when a concept is referred to by more than one term. An intelligent tool can use the knowledge base to find these alternative names and expand the search accordingly.

3. The knowledge acquisition (KA) process

In this section we will discuss the method that we have employed to elicit the knowledge regarding the software system we are studying at Mitel.

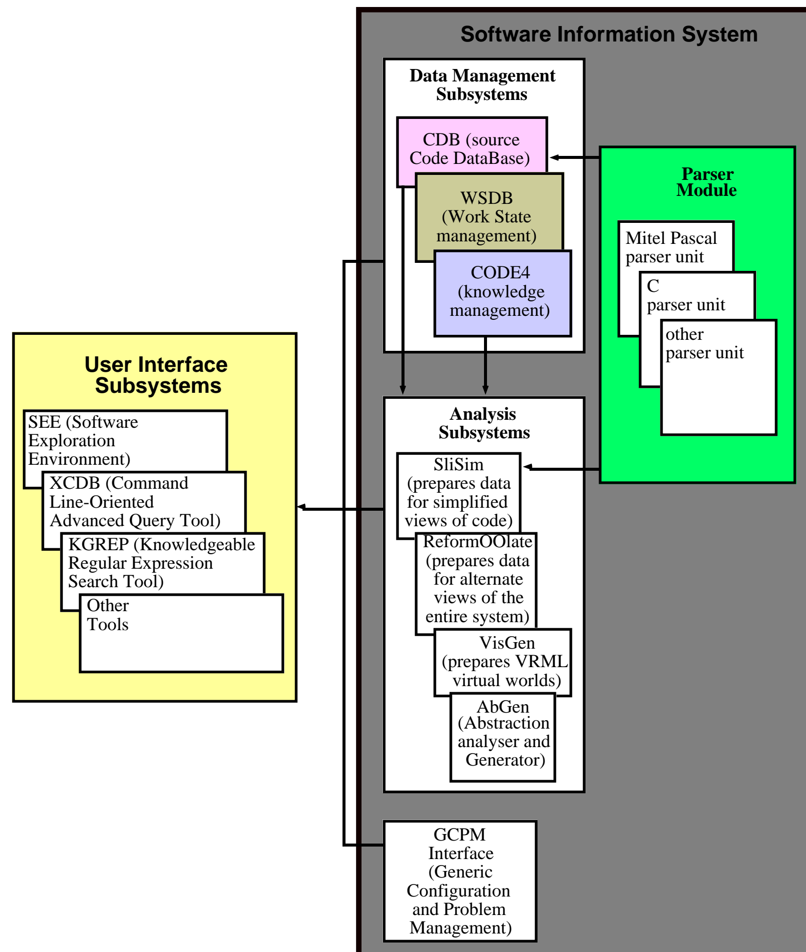

Figure 1: Overall System Architecture

From the beginning of the project we were challenged by the issue of the limitation of the SEs time. Although we credit the SEs at Mitel for their commitment and willingness to act as 'experts'[2] in the KA process, it became obvious to us that:

1) They are not going to be able to find the time to develop a knowledge base, however small, on their own; hence a knowledge engineer must be involved.
2) We must automate, as much as possible, the process of determining which concepts and relations are the most important. The involvement of the experts must be limited to responding to automated queries, and reviewing versions of the knowledge base to detect errors.

[2] In knowledge engineering, the people who provide information for the construction of a knowledge base are traditionally called the 'experts'. In our case, these experts are SEs who are performing software maintenance; many of the same SEs will also end up being *users* of the tools we develop.

3.1 Inputs to the process

We use the source code comments as the initial input to the knowledge acquisition process. We hypothesize that:

1. Almost all the important high level concepts will be mentioned in source code comments.
2. High level concepts will be among the most frequent terms in comments (if we first extract the most common English words and the names of data structures, routines and files).

Therefore, our process extracts the most frequent terms from the source code and uses these as the basis for

the knowledge base. After initial extraction of terms, the expertise of the software engineers comes into play; they are asked to pick those concepts that really are important and to organize them into a knowledge base.

3.2 Steps in the process

The following steps constitute our process. The first 6 steps are the KA phase:

1. Build a list of terms (concepts) by scanning the source code.
2. Ask the experts to rank these terms according to their importance in the context of the software component which we are studying and create a refined list of the most important concepts.
3. Ask the experts to categorize these concepts in terms of the is-a relation
4. Analyze the expert groupings to come up with an initial hierarchy of concepts
5. Refine the hierarchy using structured reviews and the system documentation.
6. Ask the experts to augment the knowledge base by supplying other relations (e.g. 'part-of', 'implements' etc.
7. Put the knowledge base into production.
8. Maintain the knowledge base by asking experts to review it periodically. Since the knowledge base is small, reviewing it in its entirety should not be time-consuming. This review process could be triggered automatically when changes are made to places in the maintained source code to which the knowledge base refers.

In what follows we will present more details about each of the above steps, and our initial experiences.

3.2.1. Creation of a concept list. We create the initial concept list by simply scanning the code comments and creating a table of terms[3].

The entries in the table are weighted according to the following empirical formula:

$$weight = W * (min(Files, W / Files)) + Files$$

in which

$W = NumOccur * Factor$

NumOccur is the number of occurrences of a term in all files

Files are the number of files in which a term has occurred at least once

Factor for a term depends on the number of words in the term as shown in the following table:

Number of words in a term	Factor
1	1
2	8
3	40

As it can be seen, the above formula gives more weight to the sequences that contain more than one word. This is to compensate for the lower probability of such sequences appearing in comparison to single words. We also assume that longer sequences of words that appear more than once have higher information content compared to single words. This method also gives more credit to terms that appear many times in many files. We believe that these terms have a higher probability of representing important concepts in the system.

We have applied the above method to a core component of our subject system at Mitel. This component contains approximately 200K line of commented code in over 160 files (‘Header’ files that contain such things as common type definitions were ignored).The resulting table contained about 13,000 terms with weights ranging from 145041 to 3.

To bring the table to a manageable size we first removed the entries that had appeared only once then browsed through the table to remove the terms that appear, with a good degree of confidence, to not represent any valuable information. Examples of these terms are variables such as i and j, terms that are the abbreviation or shorter form of other terms in the table, or English words that are not being used in a technical context. At the end we chose approximately the first 1000 highest weighed entries. The resulting table contained terms with a weight between 27 and 145041 inclusive.

The following are interesting observations about this first step:

- It is very heuristic, yet appears to do a good job of selecting important high level concepts. It certainly formed a good base for the next step, since experts did not need to change it very much.
- It is a very rapid process, taking only a few hours for a very large system (even for a knowledge engineer who has only peripheral knowledge of the system).
- It does not consume any of the experts' time.

[3] A word is a sequence of alphanumeric characters bounded by whitespace or punctuation. A term is a sequence of 1 to 3 words.

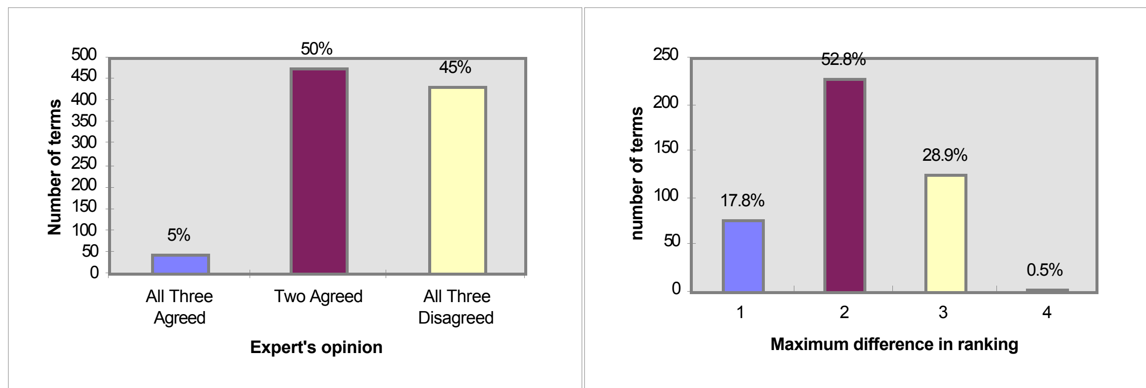

Figure 2: Differences in opinions of the experts about the importance rankings of terms. Left: The number of terms where the experts agreed or disagreed. Right: The magnitude of disagreement, among three experts, for those terms where all three disagreed.

3.2.2. Ranking the words by experts. After creating our initial word list we ask a small subset of the SEs to rank these words in terms of their importance in the context of the particular software system.

Each of the participating SEs performs the ranking individually. To facilitate this task, we provide the SEs with a simple tool which allows them to assign an importance value of 1 to 5 to each concept, 1 being not important and 5 meaning very important. The SEs also have the option of indicating that they do not know anything about a specific concept. The concepts are presented in alphabetical order so the SEs are not affected by the ranking obtained by occurrence frequency.

The interface allows the SEs to add a new concept to the existing term list if they think that it is an important concept which is missing. During the process of ranking, we maintain a minimal interference policy. We present the user interface and explain its features and what we are expecting from the SEs; then we let them to do the ranking alone.

After the ranking process, we calculate an average weight for each concept. The concepts with an average weight of 3 and higher are kept for further processing in the next stages.

In our experiment, we asked three of the Mitel software engineers to rank the concepts. The experts had different amounts of work experience with the subject software, ranging from 3 to 7 years (including the most experienced team member). After the process was complete, the number of concepts had been reduced to 403, of which 30 had been added by one or more software engineers. Each software engineer took about 1 hour to perform this step. Figure 2, describes the differences of opinions among the software engineers.

3.2.3. Categorizing concepts. In the next stage we want to see how the experts group the concepts that we generated in stage 2. The experts are provided with another tool which is shown in figure 3. This allows them to create 'groups' of concepts, and hence more general and abstract concepts. They can put any existing concept in one or more groups, and they can add whatever groups and names they desire.

There are a variety of ways to group concepts; we ask the experts to categorize the concepts given to them in terms of the is-a relation (where the group name represents the superconcept). Not all of them will be able to accurately do this, but such deviations will be dealt with later.

To keep the interface simple we do not provide facilities to create several levels of nested groups. If they want to do this, we ask them to use a period, "." to separate a category from its subcategories e.g. device.sets.display might be a low level category, with device.sets and device as higher categories.

In our experiment at Mitel, we used a different but intersecting set of experts than we used in the previous step. The idea is to maintain continuity, but also allow others to inject their ideas. The result of this process were four rather different groupings.

3.2.4. Analyzing the expert groupings. One reason to create multiple categorizations (one for each expert), as opposed to creating a single one by asking a group of experts to do the categorization together, is to capture the different ways of looking at the same data. We want to give our experts the opportunity to show us how they see the system from their own perspective. Our other alternative, i.e. categorization done by a group had the following drawbacks:

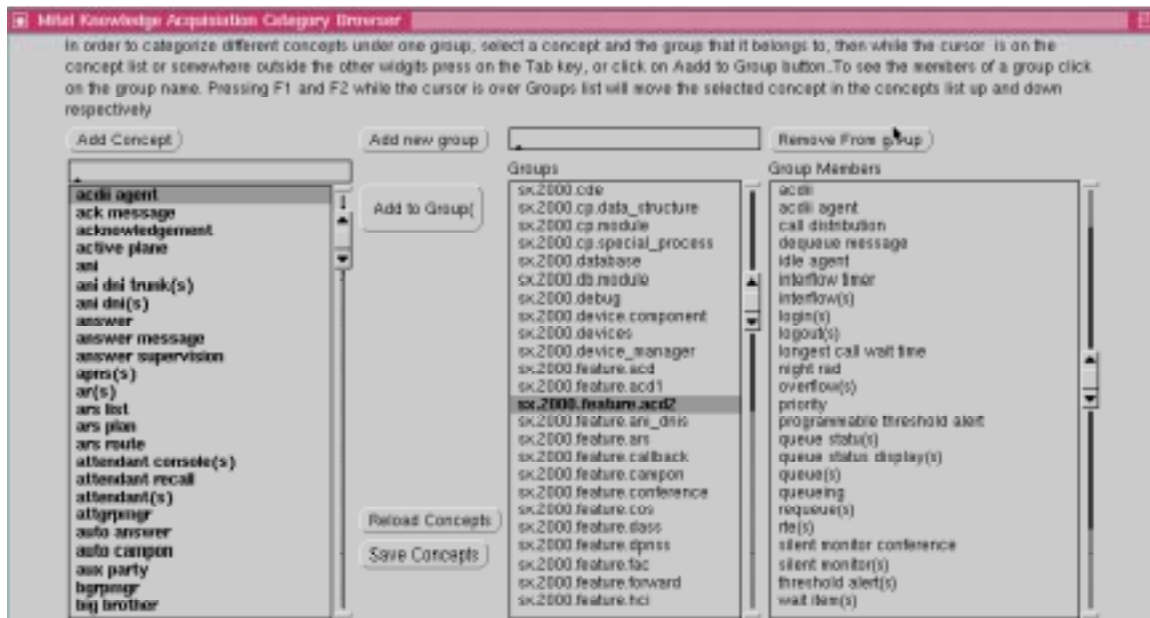

Figure 3: The user interface for the categorization tool.

- Arranging a meeting between a group of experts, with a diverse range of assignments, specially during a major release, is not an easy task.
- Junior members of the group in the presence of the more expert members may not present their views, specially when they find their view is different from the senior SE.

While it is true that more experienced SEs have a better global view of the system; we have found that different SEs tend to have a better understanding of particular subcomponents of the system, as compared with their coworkers (including the expert members of the group). In typical legacy systems there is no single person who knows each and every part of the system. Instead, people learn the details of a component of the system during working on an assignment that involves that component.

To find out the equivalent categories among different experts, we group the categories that share the same concepts together. These groupings are sorted based on the number of common concepts between categories. The higher the number of shared concepts, the stronger the possibility of equality of categories in a group. In generating the groups, our program creates the biggest possible grouping of categories.

At the end of this stage, and after discarding groups that only shared in one concept, we were left with 153 groups and 379 total concepts.

3.2.5. Creating the first version of the hierarchy of concepts. After creating the equivalent category groups, the knowledge engineer manually studies the groups to select the most appropriate name to represent a concept.

In our experiment, the experts' application of the dot notation assisted us in the creation of the original hierarchy of concepts. The resulting hierarchy of concepts was stored in a CODE4 knowledge base [7, 8]. From this point onward every change in the concept hierarchy was directly applied to the CODE4 knowledge base.

3.2.6. Refining the hierarchy by interviewing the experts. Having an initial hierarchy of concepts, we asked two of the SEs in the team to review our hierarchy together and assist us in refining it. One of the SEs was very experienced and was among the original four SEs assisting us with ranking and categorizing. The other SE had the same number of years of experience with the system but did not participate in the earlier stages of the KA process. This allowed us to benefit from the feedback and expertise of yet another SE.

We found that the mere fact of doing the interviews with two SEs at the same time initiated very interesting discussions and helped us with the better understanding of the system while giving us higher confidence in the accuracy of the generated hierarchy. As we reviewed each concept and its place in the hierarchy, we solicited ideas for other relationships among concepts as well as a simple English description for the concept.

The review process took 5 sessions, each lasting about 2 hours. After each interview we updated our hierarchy by applying the suggested changes by SEs and referring to the existing system documents. These were mostly documents that describe the functionality of the system without going into very many details of the implementation.

Having a hierarchy before starting our interviews helped us to save a large amount of time. If we had started to gather knowledge about the important concepts in the system, without the earlier semi-automatic phases, we would have had little to guide our work.

3.2.7 State of our work. In the next stage we are planning to present the resulting hierarchy to the most expert member of the team and acquire his feedback about the concept hierarchy. Our hope is to benefit from his global view of the system. Once this stage is over, the software engineers will start adding other important details about the concepts.

We plan to start experimenting with the use of the knowledge base to assist program understanding as soon as the hierarchy is complete.

4. Conclusions

We are experimenting with the use of very simple conceptual knowledge bases as an aid to program understanding. Such knowledge bases draw on technical terms found in source code comments and the combined knowledge of experts in the software system. They do not consume much of the time of busy software maintainers to build, and should be maintainable due to their small size.

Initial results show that such knowledge bases can be efficiently constructed and that they give a useful view of the system. We plan next to enhance various searching and browsing tools. Such tools will use the knowledge base to assist beginners to understand the system better and to help everybody search more effectively within the code.

Acknowledgments

We would like to thank the software engineering team at Mitel who we are using to develop and test our approach, including Steve Szeto, Soo Tung, Ian Duncan, Marc Beauregard and Glen Ala. We thank Pierre Fauvel for testing the user interfaces. Also, we thank Nicolas Anquetil and other members of our research group for their comments.

References

[1] T.C. Lethbridge and J. Singer, "Strategies for Studying Maintenance", *proc Workshop on Empirical Studies of Software*, Monterey, November 1996.

[2] P.J. Layzell, M.J. Freeman and P. Benedusi, "Improving Reverse-engineering through the Use of Multiple Knowledge Sources", *Software Maintenance* **7**, 279-299, 1995.

[3] J. Mylopoulos, A. Borgida, M. Jarke and M. Koubarakis. "Telos: a Language for Representing Knowledge about Information Systems*", ACM Transactions on Information Systems*,. **8** (4), pp. 325-362, 1990.

[4] S.R. Tilley and D.B. Smith, "Coming Attractions in Program Understanding", *CMU/SEI-96-TR-019*, Software Engineering Institute, Carnegie Mellon University.

[5] D. Skuce, "Knowledge Management in Software Design: a Tool and a Trial", *Software Engineering Journal*, Sept. 1995, pp 183-193. [6] T. C. Lethbridge, & D. Skuce, "Beyond Hypertext: Knowledge Management for the Technical Documenter" *Proc. SIGDOC 92*. Ottawa: ACM.

[7] D. Skuce and T.C. Lethbridge, "CODE4: A Unified System for Managing Conceptual Knowledge", *Int. J. Human-Computer Studies* **42** (1995), 413-451.

[8] T.C. Lethbridge, *Practical Techniques for Organizing and Measuring Knowledge*, *Ph.D. Thesis*, Department of Computer Science, University of Ottawa, 1994.