From System Comprehension to Program Comprehension

Christos Tjortjis+, Nicolas Gold+, Paul Layzell+, Keith Bennett*

+ Department of Computation, UMIST.
* Department of Computer Science, University of Durham.

Corresponding author:
Christos Tjortjis, Department of Computation, UMIST, P.O. Box 88, Manchester, M60 1QD, UK.
TEL/ FAX: +44 161 2003304/ 2003324, email: christos@co.umist.ac.uk

Index terms- Program Comprehension, Software Maintenance, Data Mining, Concept Assignment.

Abstract

Program and system comprehension are vital and expensive parts of the software maintenance process. In this paper, we discuss the need for both perspectives and describe two methods that may be integrated to provide a smooth transition in understanding from the system level approach to the program level.

The first part of this paper presents the results from a qualitative survey of expert industrial software maintainers, extracting from these results their information needs when comprehending software. Then we critically review existing software tools which can be used to facilitate system level and program comprehension.

Two successful methods from the fields of data mining and concept assignment are discussed, each addressing some of these requirements. Results from each method are given. Finally, we describe how these methods can be coupled to produce a more comprehensive software comprehension method which partly satisfies all the requirements. Future directions including the closer integration of the techniques are also identified.

1. Introduction

Software maintenance accounts for the largest cost in the software lifecycle [29]. Within the process of software maintenance, program and system comprehension play a crucial and costly role [21]. Maintainers must understand not only the localised part of a program that they need to change, but also the context within which the change takes place – system understanding. Many support
methods and tools in the field of program comprehension (the term is often applied to both program and system level comprehension) are focussed at one or the other. In this paper, we show how such methods may be coupled together to produce a more complete support environment for the software maintainer. This environment allows for switching between system and program views and partly satisfies all the requirements of industrial scale software comprehension.

The remaining of this paper is organised as follows: Section 2 presents the requirements and practices of industrial software maintainers identified by a qualitative survey conducted in the U.K. Section 3 reviews existing software comprehension tools. Sections 4 and 5 present two methods for system and program level comprehension respectively, together with results. Section 6 discusses the extent to which these two methods satisfy maintainers’ requirements. Section 7 proposes ways for combining the two methods in order to satisfy the complete set of maintainers’ requirements. The paper concludes with directions for further work introduced in Section 8.

2. Software Maintenance Requirements

Domain knowledge, understanding, and expertise are crucial factors in the software maintenance process and the type of knowledge required changes over the lifetime of the software. A clear outcome of one of the working sessions of IWPC 2000 was that there are no explicit guidelines given a program understanding task, nor are there good criteria to decide how to represent knowledge derived by and used for program understanding [1]. A fundamental research challenge therefore was to understand the key industrial needs, objectives and assumptions in the program comprehension process and to provide the most appropriate support for the specific task in hand at the time it is needed.

To determine the needs of software maintainers, understand their broad strategies, particularly the initial steps in program comprehension, and thereby provide better tool support, a qualitative survey of expert software maintainers was undertaken [33]. The survey confirmed that there is no high-quality substitute for experience when it comes to understanding and maintaining a system, as existing methods and tools are not effective enough and documentation tends to be unreliable.

The main Software Maintenance practices and requirements identified by this survey were the following:

1. High level overviews, abstractions, sequence and localised diagrams of the system, module interrelationships and also means to estimate the impact of changes are required to be derived in an automated or at least semi-automated manner in order to accelerate and enhance program comprehension.
2. It was reported that mental models of programs, i.e. high level abstractions of subsystems with related functionality and interrelationships, are implicit in maintainers’ work, but are hardly ever recorded for future use. The need for visualising, recording and cross-referencing these models in order to share understanding or experiences, improve communication and resolve misunderstandings was clearly identified.

3. Identification of a starting point for subsequent searching and tracing through programs significantly accelerates the comprehension process. This normally occurs through consultation with experts in the system and by use of maintainer’s own experience. Alternative means to locate a starting point are essential.

4. Information exchange among team members is sparse, informal and is hardly ever formally recorded. There are no effective mechanisms to share understanding or experiences. There is a clear requirement for a means to provide standardised, reliable and communicable information regarding a software system as an equivalent to knowledge available only to developers or experienced maintainers.

5. Maintenance activities are mainly documented in source code comments, except from extensive changes which are also reflected on user manuals. These inline comments detail the reasons for changes and the way they were implemented. The implication is that comments in mature systems get accumulated over time and tend to reflect subsequent changes rather than the original implementation ideas. Capturing knowledge regarding past modifications by extracting information from comments and relating this to known functionality of code emerges to be of great importance.

6. The types of maintenance were found to influence the approach taken. Corrections normally involve attempting to locate the point where the fix needs to be applied, as soon as possible. Enhancements usually require a ‘detail-first’ strategy, where a high-level understanding of the system’s functionality and modules interrelationships is pursued before the change is made, before assessing the wider impact of the change using techniques such as regression testing. Preventative maintenance was deemed rarely to occur and was considered to be an integral part of software development. The above highlight the fact that maintainers are often required to be able to switch between System Level Comprehension and Program Comprehension.

7. Partial program comprehension is pursued and achieved in most cases, which has to be balanced against the risk of failure in completing a maintenance task. It was reported that the time available for program comprehension was limited because of commercial pressures and delivery deadlines.

It was generally agreed that the most useful pieces of information to facilitate code comprehension when maintaining software are:
a. An easy to navigate multi-layered subsystem abstraction, capturing control flows and modules interrelationships providing an overview of the system and possible impact of changes.

b. Knowledge derived from past maintenance which can mainly be retrieved from comments.

3. Comprehension Support

There are many types of software tool available to help with software comprehension, emphasising different aspects of software systems and modules, and usually creating new representations for them [9]. Biggerstaff et al. differentiate between naïve and intelligent agents (tools) for providing such representations [3]. Naïve agents generally perform deductive or algorithmic analysis of program properties or structure, e.g. program slicers (see [30]) or dominance tree analysers (see [5]). Intelligent agents attempt to assign descriptions of computational intent to source code.

Biggerstaff et al. [3] claim that research on intelligent agents can be divided into 3 distinct approaches:

1) Highly domain specific, model driven, rule-based question answering systems that depend on a manually populated database describing the software system. This approach is typified by the Lassie system [7].

2) Plan driven, algorithmic program understanders or recognisers. Two examples of this type are the Programmer’s Apprentice [26], and GRASPR [37].

3) Model driven, plausible reasoning understanders. Examples of this type include DM-TAO [3], [4], IRENE [19], and HB-CA [9], [11].

One exception to this categorisation is Hartman’s work [13] that falls between approaches 2 and 3.

Systems using approaches 1 and 2 are good at completely deriving concepts within small-scale programs but cannot deal with large-scale programs due to overwhelming computational growth. Approach 3 systems can easily handle large-scale programs since their computational growth appears to be linear in the length of the program under analysis. They suffer from approximate and imprecise results [3].

Figure 1 is based on the summary of the program understanding landscape in [3] as extended in [9]. The original has been updated to include additional work on program understanding, with the number of each oval providing a key to the citations below. Citations have also been added to the original figure.
Key to citations

<table>
<thead>
<tr>
<th>Oval</th>
<th>Author(s)</th>
<th>System</th>
<th>Citation(s)</th>
<th>Oval</th>
<th>Author(s)</th>
<th>System</th>
<th>Citation(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Karakostas IRENE</td>
<td>[19]</td>
<td></td>
<td>4</td>
<td>Wills GRASPR</td>
<td>[26], [36], [37], [38]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>Kozaczynski Concept</td>
<td>Kozaczynski [20]</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Gold</td>
<td>HB-CAS</td>
<td>[9], [10], [11]</td>
<td>4</td>
<td>Johnson PROUST</td>
<td>[17], [18]</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Rich, Waters</td>
<td>Programmer’s Apprentice</td>
<td>[34], [35], [25], [26], [27], [28]</td>
<td>4</td>
<td>Chin, Quilici DECODE</td>
<td>[6]</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Woods et al.</td>
<td>PU-CSP</td>
<td>[22], [39], [40], [23], [43], [24], [41], [42]</td>
<td>4</td>
<td>Harandi PAT</td>
<td>[12]</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Hartman</td>
<td>UNPROG</td>
<td>[13], [14], [15]</td>
<td>5</td>
<td>Biggerstaff et al. DESIRE</td>
<td>[2], [3], [4]</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: The Program Understanding Landscape [9] after [3]

4. A Method for System Level Comprehension

Data mining involves applying data analysis and discovery algorithms to data collections that, under acceptable computational efficiency limitations, produce a particular enumeration of patterns over the data [8]. Data mining incorporates several techniques used to get insight into vast amounts of data and extract useful, previously hidden knowledge.
**Clustering** is one of these techniques employed for partitioning a data set into mutually exclusive groups (clusters). Members of a cluster should be similar to one another and dissimilar from members of other groups, according to some metric. Similarity is decided by measuring the distance of records with respect to all available variables [16].

**Data Mining Code Clustering (DMCC)** [32] is an approach, devised to address the need for automated methods providing a quick, rough grasp of a software system, to enable practitioners, who are not familiar with it, to commence maintenance with a level of confidence as if they had this familiarity. This “rough-cut” approach to program comprehension places emphasis on supporting the maintainer sufficiently to start a task, with a tool providing the equivalent of an inexperienced maintainer consulting with an experienced maintainer in order to scope a problem and get started.

DMCC primarily aims at providing a broad contextual picture of the system under consideration, rather than a refined, detailed model [32]. Such a broad model provides a basic roadmap by which maintainers, who lack a detailed knowledge of a system, can navigate around the code, scoping the change request and solution space in a relatively short period. This in turn enables more detailed analysis of targeted code to be undertaken, minimising analysis and computation time.

DMCC involves representing a program as a number of entities that are grouped in clusters representing subsystems, based on their similarity. These clusters indicate structure amongst functions and also interrelationships between them, in a way that the impact of changes can be predicted with an acceptable amount of uncertainty. Central issues for DMCC were the specification of program entities and their attributes, similarity metrics, and clustering strategy. A prototype tool for clustering C/C++ source code was developed, using functions as entities. Attributes employed by DMCC are the use and types of variables / parameters and the types of returned values. Additional information about interrelationships amongst attributes is also fed to the tool. Custom-made similarity metrics based on the association coefficient paradigm, were introduced and an agglomerative hierarchical clustering algorithm using the complete linkage method was employed.

The tool was evaluated using input data extracted from C/C++ systems of various sizes. Experimental results indicate that a high-level system abstraction as a number of subsystems can be achieved by clustering program functions into groups depending on the use and types of parameters. Interrelationships amongst components were identified in a similar manner. The accuracy of the results was evaluated by comparing the produced subsystem abstractions with experts’ mental models where available. The abstractions were accurate, capturing the subsystems consistently with the mental model. Pair-wise values of precision and recall ranged between (50%, 40%) and (87%, 100%), i.e. highest precision achieved was 87% and highest recall 100% [31].

Grouping program components into subsystems reduces the perceived complexity thus facilitating maintenance. Corrective and adaptive maintenance is supported by the automatic derivation of a
meaningful decomposition of source code into several subsystems, by identifying the interfaces between subsystems and determining the role each plays in performing a service [32]. This can further help to modify existing code in a manner consistent with the original structure and understand the overall impact of such modifications. Any changes, especially those related to parameter usage within the body of a function, suggest the maintainer should consider the possibility of other “similar” functions being affected. This supports fast code modification risk assessment, before even performing regression tests which in practise are time consuming and often neglected. Maintainers should even be enabled to replace code sections of code without affecting functionality.

DMCC can also be used for perfective maintenance, when attempting to improve systems cohesion and coherence by increasing modularity. This could happen in two ways. Firstly, functions could be relocated within modules where they more “naturally” belong. Secondly, processing within functions could be adjusted to better reflect the functionality that is supposed to be encapsulated within.

5. A Method for Program Level Comprehension

Concept assignment is a process aimed at assisting the maintainer in program comprehension by indicating where operations (e.g. Read) or entities (e.g. File) exist within the source code. It involves identifying the location, scope, and instance of concepts within source code. The type of concept assignment we are concerned with in this paper is termed plausible-reasoning owing to its use of multiple information sources (including informal clues such as comments) to assess the likelihood of the occurrence of a concept in the code. This approach differs from the common alternative of deriving the concepts from the semantics of the programming language (see section 3). The advantage of plausible-reasoning systems is their scalability over any size of program.

The Hypothesis-Based Concept Assignment (HB-CA) method [9], [10], [11] is a plausible-reasoning technique for identifying abstractions and concepts in COBOL source code. Concepts are proposed by a maintainer and stored in a library as simple text strings. They are classified as either actions (i.e. they do something) or objects (they are something on which actions take place). Each concept has one or more indicators (also text strings) that, when found in source code, may indicate the presence of the particular concept. Indicators are assigned to different classes: identifier (variable/procedure names), keyword, and comment (single words only, no phrases). Concepts can be joined by specialisation (one object to another) or in composition (one action with one object). An example might be the Read concept. It is an action and may have a single comment indicator of READ. It could be combined with the File concept (an object) which itself may be specialised by the
MasterFile concept. The composition would form the Read:File concept. Once a basic library has been defined, the method can be applied.

HB-CA is a three stage method comprising Hypothesis Generation, Segmentation, and Concept Binding. The library is used by the Hypothesis Generation stage to analyse the code and produce hypotheses for every concept whose indicators are found. Indicators can be matched directly or with flexibility (e.g. using sub-strings or synonyms). The resulting hypothesis list is passed to the Segmentation stage. This attempts to group hypotheses into coherent segments, focussed around single concept. It undertakes this by using the subroutine boundaries present in the original source code. Where the code has no subroutines or they are very large, an unsupervised neural network is used to learn the conceptual structure of the hypotheses being considered and smaller segments defined based on this analysis. The segments are passed to the final stage: Concept Binding. This uses the weight of evidence for a concept (in terms of number of hypotheses) to determine which concept is dominant and thus present in the segment. If several concepts have the same level of evidence, a number of disambiguation rules are applied to pick a winner. The output is shown by colouring portions of the source code to match a coloured concept name displayed next to the code.

HB-CA has been evaluated using 22 real-world COBOL II programs and an example library of 23 concepts, each having one or more indicators. The results are promising, showing a mean accuracy of between 56% and 88% depending on the options used. Its computational growth is linear in the length of the source code being analysed, and the knowledge base can be updated with minimal effort.

6. Satisfying the Needs of Software Maintainers

As explained in section 2, despite existing methods and tools for system level and program comprehension, practitioners in the industry impose a set of requirements yet to be satisfied. Section 4 and 5 respectively introduced two methods, namely Data Mining Code Clustering and Hypothesis-Based Concept Assignment, facilitating these types of comprehension. This section presents the way these methods individually address most of the above requirements. Furthermore, we discuss how coupling of the two methods can satisfy the remaining requirements, making this combination a potentially complete answer to each one of the specified maintainers’ needs.

DMCC is an approach which successfully addresses the first two requirements set by the industry. More specifically it produces a high level overview of a system, where modules are grouped together according to their similarity and their interrelationships are highlighted. It also provides the means to visualise and record a representation of a system, resembling a mental model which can be used to confirm perceptions, communicate these models and cross-reference them across a team. DMCC also
provides maintainers with the required multi-layered subsystem abstraction which captures module interrelationships and can indicate the possible impacts of modifications.

HB-CA successfully addresses requirements 2, 3, 4, and 5. The need to share mental models is facilitated to some extent by the use and extension of the knowledge base by several maintainers. HB-CA provides a particularly good method for identifying the starting point for maintenance by providing the maintainer with a program representation in conceptual terms that they have nominated. The starting point can be expressed in terms closer to the problem. The shared knowledge base enables the recording of knowledge highlighted in requirement 4. Although the knowledge base structure is not elaborate, it does provide a mechanism by which maintainers can store parts of their system and program understanding for others to use. One of the main sources of knowledge for the HB-CA analysis is inline comments, and it uses these to determine which concepts are implemented in a particular section of code. It can be seen as a knowledge capturing method as desired in requirement 5.

The result of coupling DMCC and HB-CA addresses the rest of the requirements set by industrial practitioners, i.e. switching between System Level Comprehension and Program Comprehension (requirement 6) and accelerating and improving the quality of partial comprehension (requirement 7). The way these further requirements are met will be explained in the following section.

7. Combined Method for Better Support

This section describes ways in which DMCC and HB-CA could be combined to improve the support offered to software maintainers.

Data mining gives an overview of the interrelations among low-level modules (functions) found in program files. Therefore:

- It can be used to assess modularity.
- It may be used for code ripple analysis and risk/impact analysis.
- It could be used prior to remodularization.

Concept assignment gives an overview of the concepts found in a particular program file by mapping concepts (terms) to their implementation in code. Therefore:

- It can be used for business rule/code ripple analysis and risk/impact analysis.
- It can be used for module selection prior to change.
- It can be used to help with code reuse.
It’s useful in software module comprehension

There are several ways in which data mining could be coupled with concept assignment to improve the completeness of comprehension support:

a. DMCC could assist in CA knowledge base generation. DMCC could be used to locate indicators (perhaps within the data sections of programs) and possibly concept-concept relationships. Concepts produced by DMCC are of “higher order” than the ones usually stored in the knowledge base. For example, instead of having a read concept, DMCC can introduce a sort concept which in fact consists of concepts of “lower order” such as read, write etc. This hierarchical approach extends the scope and enriches the usefulness of CA.

b. Segmentation could be based on DMCC “clusters” rather than regions of code formed between primary segmentation points or as an alternative to using neural network processing to find conceptual coherence. HB-CA initially segments code at section boundaries and then by use of Self-Organising Maps (SOMs) to reflect the conceptual structure of the program as expressed in terms of the knowledge base content. DMCC suggests further groupings of routines or paragraphs, which are more likely to contain “higher order” concepts and relationships.

c. Enhanced code ripple analysis and module selection (i.e. what’s the change and what is affected?) As both DMCC and CA may be used for code ripple analysis and risk/impact analysis results can be cross-validated when “overlapping” or combined when addressing different issues.

d. Cross-validation of DMCC and CA findings. This may happen if, instead of coupling the processes of the two methods, we only allow their results to be coupled. In other words, as DMCC produces high-level results and HB-CA produces low-level ones, there is a valid expectation that these can complement each other. This can be achieved by highlighting different aspects of a system or by providing two different angles for viewing a single aspect, lying in the boundaries of the scope of each method.

8. Conclusions and Future Work

System and program level comprehension is crucial for industrial scale software maintenance. A set of relevant requirements identified during a survey is only partly met by existing methods and tools. In this paper we have presented two methods that meet most of these needs individually. We have also proposed several ways in which they may be combined to greater effect and to provide more substantial support. This combination potentially addresses all the requirements.

There are a number of directions for further work in this area:
1) **Empirical validation of the combined approach.** It would be interesting and useful to expose the combined method to maintainers in the real world to determine whether it can actually meet the needs identified in the early part of this paper.

2) **Closer integration between the methods.** The current style of coupling between the methods is loose and maintainers would benefit from a closer fit between them, as it would give them the ability to switch quickly between system views.

3) **Framework Development.** Many aspects of data mining are adopted in program comprehension tools and we plan to develop a framework to characterise and classify such tools by the data mining methods they adopt for data extraction and processing.

**Acknowledgements**

We gratefully acknowledge the support of EPSRC, the Leverhulme Trust, and CSC for various aspects of this work.

**References**


