IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Suite 200, Hershey PA 17033, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP4592

Recreating Design Artefacts of Information Systems for Systems Evolution and Maintenance

Khaled Md. Khan
University of Western Sydney, Locked Bag 1797, Penrith South DC, 1719, NSW, Australia,
Tel: +61-2-9685-9558, Fax: +61-2-9685-9245, k.khan@uws.edu.au

Yi-chen Lan
University of Western Sydney, Locked Bag 1797, Penrith South DC, 1719, NSW, Australia,
Tel: +61-2-9685-9283, Fax: +61-2-9685-9245, yichen@cit.uws.edu.au

ABSTRACT

The paper presents an overview of the process of understanding existing information systems in order to aid system evolution and maintenance. The outcome of the paper is largely based on the experiences gather from a system maintenance project. The paper argues that the process of understanding existing systems involves four major design abstraction levels. To achieve the highest level of design abstraction in the hierarchy, maintainer programmers have to follow a defined process to gain a complete understanding of the system structure and semantics.

INTRODUCTION

Information systems (IS) always tend to change and evolve as technology and business rules change. Evolution of information systems is unavoidable, and it is a natural phenomenon. Organizations need to support systems evolution to take advantage of the new technology and to address the changing business rules. In the era of web-based systems and e-business, organizations need appropriate maintenance process and resources that are required to migrate their aging legacy information systems to web-enabled contemporary systems. The need for a system evolution emerges from various issues such as changes of business rules, emerging new technology, need for new functionality, or fixing defects in the systems and so on.

A major component of system evolution is to comprehend the underlying design rationale of IS at various levels. This paper explores issues of recreating design artefacts at various levels of abstractions. Current practices of system comprehension activities revolve around ad hoc patching which do not follow any defined methodology. A more defined formalism describing various levels of design artefacts and the abstraction process to recreate the underlying design knowledge is required to enable maintainers a clear understanding of the system.

The motivation of the work reported in this paper was actually generated from a maintenance project of a business application system. The candidate system was a small Inter Bank Reconciliation System (IBRS) used in a developing country in Asia. The organization later decided to transform the system into a more portable and efficient programming language platform keeping the entire functionality of the system intact. One of the authors of this paper was assigned the responsibility to lead the project. It involved considerable re-engineering task. The maintenance experience with the project reported in (Khan *et al*,. 1996; Khan and Skramstad, 2000; Khan et al., 2001) has motivated us to propose a program understanding process in this paper.

The paper proceeds as follows. In the next section, we outline the experiences with the maintenance process. We describe the issues related to program understanding process in section 3. The paper concludes in section 4.

EXPERIENCE WITH THE MAINTEANCE PROJECT

Beginning of the maintenance project with IBRS, it was learned that no design documents of the candidate system were produced during the development process. The system did not follow any coding standard, and no design documentation was available. One of the original programmers involved in the development of the system provided several informal diagrams about the system dependency. We combined the information gathered from her with the informal scenario that we already obtained to grasp the overall structure of the system. We tried to trace manually the flow of execution of the system to keep track of each function as reported in Khan et al., 2001.

First mental representation of the system we built was a program model as defined in (Pennington 1987; Mayrhauser *et al.*1994) by identifying the flow of control structures, call sequences, and scopes of global and shared variables found in the source code. We tried to map the relationships among all the scattered programming elements. This mapping process later allowed us to reconstitute the fundamental architecture of the system as well as the interaction of various programming components such as global and shared variables, function calls and branching structures.

PROGRAM UNDERSTANDING PROCESS

Based on the experience with the maintenance project, we have learned several lessons in program understanding process:

(a) Two types of *systems knowledge* are required to effectively understand an IS
(b) System knowledge can be recreated at four different *levels of abstractions*
(c) Maintainer programmers need to adapt defined *design recovery process.*

Each of the above three is discussed in the subsequent sections

System knowledge

In general terms, there are two major types of system knowledge found in a system: (1) syntactic knowledge (structure of the systems); and (2) semantic knowledge (program functionality).

Syntactic knowledge

This type of knowledge is the basic building block that is extractable from the source code. The syntactic knowledge includes for example, control structures, hierarchy of calling structures, programming patters, data structures. This type of information is very much programming language dependent.

Semantic knowledge

This type of knowledge comprises the ultimate intentions of program, and domain knowledge of the entire information system. This type of informa-

tion is not always directly found in the source code and more difficult to be verified its correctness. We discuss the importance of informal linguistic information in understanding the semantics of a system in the latter sections.

Levels of abstractions

A system can appear to be in different levels of abstraction at different stages of its existence (Hausler et al., 1990). It is noted that the program understanding process gradually crosses different abstraction levels from implementation towards higher levels (Harandi and Ning, 1990). We can view a system conceptually at different levels in its existence. Program understanding is a knowledge intensive activity, and abstracting a higher level of representation requires abstracting of immediate lower level representation of the system in the hierarchy.

Figure 1 shows the hierarchical abstraction of an information system at four different levels such as: (1) implementation level, (2) structural level, (3) logical level, and (4) conceptual level.

Implementation level

To understand a system into its implementation level, one must be familiar with its language syntax and semantics. It can be derived in terms of an abstract syntax tree and a collection of program tokens. It is the lowest level of the system abstraction hierarchy.

Structural level

This level represents the dependencies among the program's different components. Control flow diagrams, control dependencies, procedure calls relationships, and structure charts are the examples of the structural level documentation. The syntactic knowledge specified in the previous section falls in this level.

Logical level

Logical level represents the logical relationship among the various components of a system. In program understanding, it is important to understand what function is provided by which part of the program. This level can be represented as data flow and data definition graphs.

Conceptual level

The conceptual level focuses on the application domain of the system. It includes the problem being solved, business rules, users' understanding of the problems and so on.

Design recovery process

Design recovery process has two phases: recreating syntactical knowledge from the source code; and recreating semantic knowledge. Maintainers first need to search for larger structural components of the system such as the subsystem structure, fundamental data structures, and module structures based on the analysis of source code and available design documents. Getting a conceptual view (a mental model of a program) of a system requires a representation of a program not as a text file but as a set of interrelated concepts. Syntactical knowledge provides maintainers enough information to recognize semantic knowledge of the system. Figure 2 shows the process on how the abstraction at the conceptual level can be achieved.

Recreating syntactical knowledge

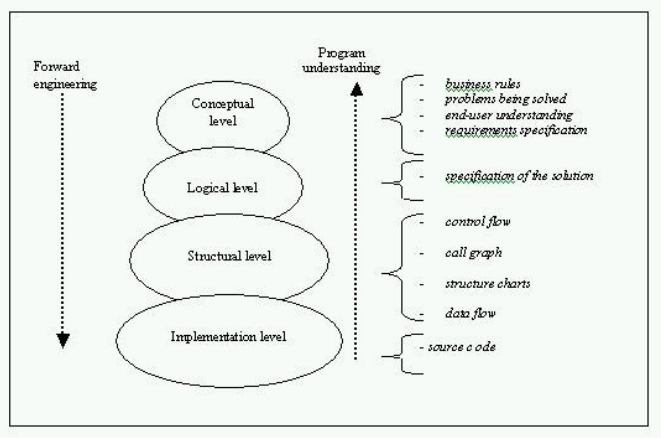
The phase of recreating syntactical knowledge requires followings:

i) Identifying the application domain
ii) Understanding dynamic behaviour
iii) Composing the system into structural representation
iv) Collecting informal linguistic information.

Identifying the application domain

Identifying the candidate system domain is the first step in system maintenance. Types of functionality and the profile of the environments in where the system being used indicates the application domain. These two ingredients are vital to perceive the system's application domain.

Figure 1: Hierarchy of program abstraction levels



Understanding dynamic behaviour

The most important stage of recreating the system design is the comprehension of system functionality. The functionality of a program needs to be understood before attempting to extract the internal mechanisms of the system. This can be achieved by running the program with real data.

Composing the system into structural representation

It is important to compose the program into larger logical units. For languages which do not support the notion of module structure, the maintainer must depend on their intuition, experience and design documents to establish a conceptual boundary of the larger program structure. It is assumed that the functions in a source code file are semantically related, and constitutes a cohesive logical module (Choi and Sacchi, 1990). It is customized that program developers generally group related functions in one source file (Choi, 1989). The existing module structures could be constituted into larger chunk of the system structure to represent a wholeness of the related system functionalities. In this regard, program slicing is a well known technique that utilizes the properties of control flow to locate all instructions sets wherever in the control flow path that invoked an event across the module boundaries. This technique of slicing isolates individual computation threads within a program.

Collecting informal semantics information

It is quite useful to take into consideration the informal information structures scattered in the source code. Some natural language texts used in the source code as comments could be used to fill the gap between the conceptual level and the implementation level of the system. IS maintainers can retrieve a wide varieties of information from the combination of informal and formal linguistics structures in the source code. This type of information is helpful to identify the semantic knowledge of the program. The informal linguistic structures in the code such as comments, naming style and convention of data structures and functions provide vital information on the actual purposes of the data and function definitions.

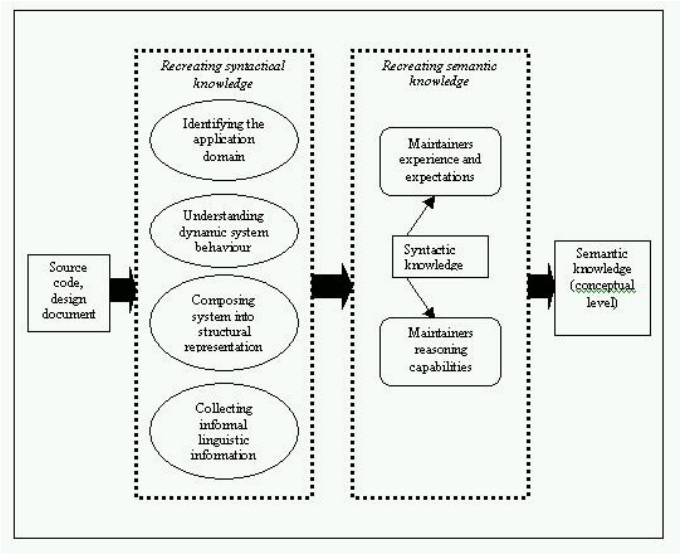
Recreating semantic knowledge

Once the syntactic knowledge is created, maintainers experience and reasoning capabilities are used as an aid to recreate the semantic knowledge of the system. This practice heavily depends on the completeness of the syntactical knowledge recreated, programmers' experience, intelligent guessing, and their reasoning capabilities. Once semantic knowledge is reproduced, it is believed that maintainers' understandings reach at the conceptual level.

CONCLUSION

This paper focuses on the process of recreating the design rationale of existing IS. We have argued that system understanding is a pre-requisite for information systems evolution and maintenance. A system understanding process requires abstraction of system knowledge at various levels in the abstrac-

Figure 2: Design recovery process



tion hierarchy. In an effective maintenance process, maintainer's understanding should reach at the conceptual level of the system knowledge. The paper also cites a design recovery framework to recreate the conceptual level of understanding of the system.

REFERENCES

Choi, S., Sacchi, W., (1990). "Extracting the Design of large Systems", IEEE Software, January, pp. 66-71.

Choi, S. (1989). "Softman: An Environment Supporting the Engineering and Reverse Engineering of Huge Software Systems", University of Southern California, Los Angeles 1989.

Hausler, F., el al., (1990), Using Function Abstraction to Understand Program Behaviour", IEEE Software, January 1990.

Khan, M. K., Rashid, M. A. and Lo, W. N. B. (1996). 'A Task-Oriented Software Maintenance Model', *Malaysian Journal of Computer Science*, Vol. 2, December 1996, 36-42.

Khan, K., Lo, B., and Skramstad, T. (2001). Tasks and Methods for Software Maintenance: A process oriented framework. Australian Journal of Information systems, Nol. 9., no. 1, September, pp. 51-60.

Khan, K., and Skramstad, T. (2000). Software Clinic: A Different View of Softare Maintenance. International Conf. On Information systems analysis and synthesis, Orlando, pp. 508-513.

Mayrhauser, A. von, Vans, A. M. (1994). 'Comprehension Processes During Large Scale Maintenance', *IEEE Proceedings Conference on Software Engineering*, 1994, 39- 48.

Pennington, N. (1987). 'Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs', *Cognitive Psychology*, Vol. 19, 1987, 295-341.