Software Clinic:
A Different View of Software Maintenance Process

Md. Khaled Khan
Peninsula School of Computing and Information Technology
Monash University,
McMahons Road, Frankston, VIC 3199
Australia.
Fax: +61-3-99044124
Email: khaled.khan@infotech.monash.edu.au

Torbjorn Skramstad
Department of Informatics
Norwegian University of Science and Technology
N-7491 Dragvoll, Trondheim,
Norway
Fax: +47 73 591733
Email: torbjorn.skramstad@ifi.ntnu.no

Abstract: The increasing size and complexity of many software systems requires a greater emphasis on an effective software maintenance environment. The activities of the maintenance process should be well viewed and defined. To support this notion we have proposed a maintenance process view called software clinic in this paper. This approach has been proposed as an improvement over current ad hoc practices in software maintenance. In particular, software clinic model has emerged in an attempt to provide an environment for software maintenance process. The motivation for proposing this metaphor has actually generated from a maintenance project of a PC based application software. The approach was the ultimate outcome of the experiences gathered from the project, not the other way around. The project was not conducted to validate and test the software clinic approach in a real life setting rather the maintenance project itself has motivated the author to propose the underlying concepts. The maintenance process view cited in this paper is considered a conceptual framework derived from a maintenance experience. The experience on software maintenance has also been described in this paper.

Keywords: Software maintenance, program understanding, reverse engineering, design artifacts.

1 - INTRODUCTION

According to the IEEE standard (ANSI/IEEE 1983) the definition of software maintenance is the "modification of a software product after delivery to correct faults, to improve performance, or to adapt the product to a changed environment". The bottom line of this definition is the "software product after delivery", not during the production time. Any modification or redesign of a system carried out during the development stage cannot be considered as software maintenance. Many software maintenance definitions have been proposed so far, and most of them are consistent with the IEEE standard.
The magnitude of software maintenance work is much higher than the development as reported by various papers presented in conferences on software maintenance sponsored by IEEE Computer Society. The tasks of software maintenance are considered difficult and complex as widely agreed by the practitioners and academics reported in various forums. Virtually, no complete maintenance environment model including its underlying activities, methods and tools used by the process have been proposed to satisfy the needs. Some scattered guidelines are available for the software managers to enable them to tailor process models to maintenance projects. In addition, management does not always give much importance to a maintenance project, and this attitude can be well justified by a lack of adequate measures, structured planning and procedure of maintenance process (Abran *et al.*, 1995).
To support various maintenance components, their activities, associated methods and tools which are considered vital for software maintenance process, we have viewed software maintenance in a different way in this paper, and termed it Software Clinic. The motivation for proposing this model was actually generated from a maintenance project of a PC based application software. The approach was the outcome of the experiences gathered from the project. The project was not carried out to validate and test the software clinic approach in a real life setting rather the project experience itself has been a motivating factor for us to propose this approach. This approach is considered as a different

view of current ad hoc practices in software maintenance. In particular, software clinic model has emerged in an attempt to view software maintenance in a different way than it is usually done. We also believe that the nature and activities of software maintenance process essentially do differ from that of development process (Skramstad *et al.* 1992; Khan *et al.* 1996). In this paper, the word 'approach' and 'model' are used interchangeably to refer the same thing.
Various aspects of our work are presented in this paper as follows. In section 2 we have cited the background of our work. Section 3 describes our experience of a maintenance project. Lessons learned from the project are outlined in section 4 which motivated us to propose the software clinic approach in section 5. A brief summary of related work is cited in section 6, and finally, section 7 indicates the direction of further work of our approach.

2 BACKGROUND OF THE WORK

Most personal computer (PC) based systems today are found mainly in small or medium size companies. Most of these are special-purpose systems originally designed to reside and operate on PCs. There is a popular trend nowadays to migrate these PC based systems to more efficient programming languages to get the full benefits of modern powerful PCs. The characteristic of the project reported in this paper falls in this category. Our candidate system was a small Inter Bank Reconciliation System (IBRS) used in a developing country in Asia. A prominent software company in Asia locally developed the entire system. The company has been long reputed for its pioneer role in producing several application software used in the region.
Our candidate system was relatively small in terms of lines of code (LOC), and was written in a fourth generation language (4GL). The system was organized hierarchically included more than 100 functions and almost 35 data files in various format. The system was capable of providing at least 40 various types of services to the user. The company that developed the system later found that the system was not portable, and it was not speed wise efficient. The management wondered whether it was yet possible to transform the system into a more portable and efficient programming language platform like C keeping the entire system functionality intact. One of the authors of this paper was assigned the job to look into the project. The author was then a senior programmer of the company. He realized that it was clearly a reengineering task. He has had a good research experience in software maintenance from Europe. As an experimental basis an intensive and systematic working environment was created for this maintenance project.

3 MAINTENANCE PROJECT

The application system was successfully transformed into C language platform within a short time. It was possible because the total LOC of the system was relatively small.
At the very beginning of the project, the candidate software was treated from an entirely different point of view. The system was considered as an object that required certain "*health care*', a term used in (Ash *et al.* 1994), and the '*entire body of the system*' will be transformed into a different programming language platform with a '*good health*'. This attitude towards the system has later prompted us to define a software clinic model presented in this paper. The entire maintenance project activity was not aided with any automated tools. A summary of the project experience has been cited in Figure 1. The various tasks performed and methods used during this project are briefly described below.
<u>Domain knowledge</u>: At the beginning of the project, the author was briefed about the application domain of the system very informal way. An informal scenario of available functions was described from the user's point of view. In fact, the domain scenario included three vital information: what services the system offered, under which environments the system was being used, and who were the frequent users. It was felt that the information was very important to perceive the system's application domain.
<u>Program understanding</u>: The domain knowledge of the system received at the early stage of the project assisted us a great deal in program comprehension process. Program understanding was considered as the '*diagnosis of the system body*'. It was realized at the beginning of the project that understanding the system behavior is the starting point for the maintenance process, and it is one of the difficult tasks in maintenance if appropriate documents were not available. It was learned that no design documents

as well as no requirement specifications of our candidate system were recorded during the development time. The system did not follow any sort of coding standard, and not a single page of documentation was available. One of the original programmers involved in the development of this system was available for consultations during the early stage of the project. Unfortunately, she was not able to recall most of the design rationale of the system, however, she provided some specific types of information about the system, like data structures, call structures and the data-file format. She drew several informal sketches to express the system dependency because she was completely unaware of the design conventions used in software engineering. 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.

We executed the system a number of times to study its behavior. When we executed the entire program with real data, the system gradually exposed its main features. Running the system with real data input was virtually forcing us to grasp the domain of the system as well as the services it provided. However, we obtained a better understanding of the code later from code walkthroughs. We realized that we had to extract syntactic knowledge from the source code to form semantic abstraction of the system. First mental representation we built was a program model as defined in (Pennington 1987; Mayrhauser *et al.* 1994) by tracking the flow of control structures, call sequences, and scopes of global and shared variables found in the source code. We tried to find out the architectural description as a collection of various programming components, which described the interactions among them. This understanding process later allowed us to identify 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. All program information obtained from the program understanding process was clearly documented in a structure format.

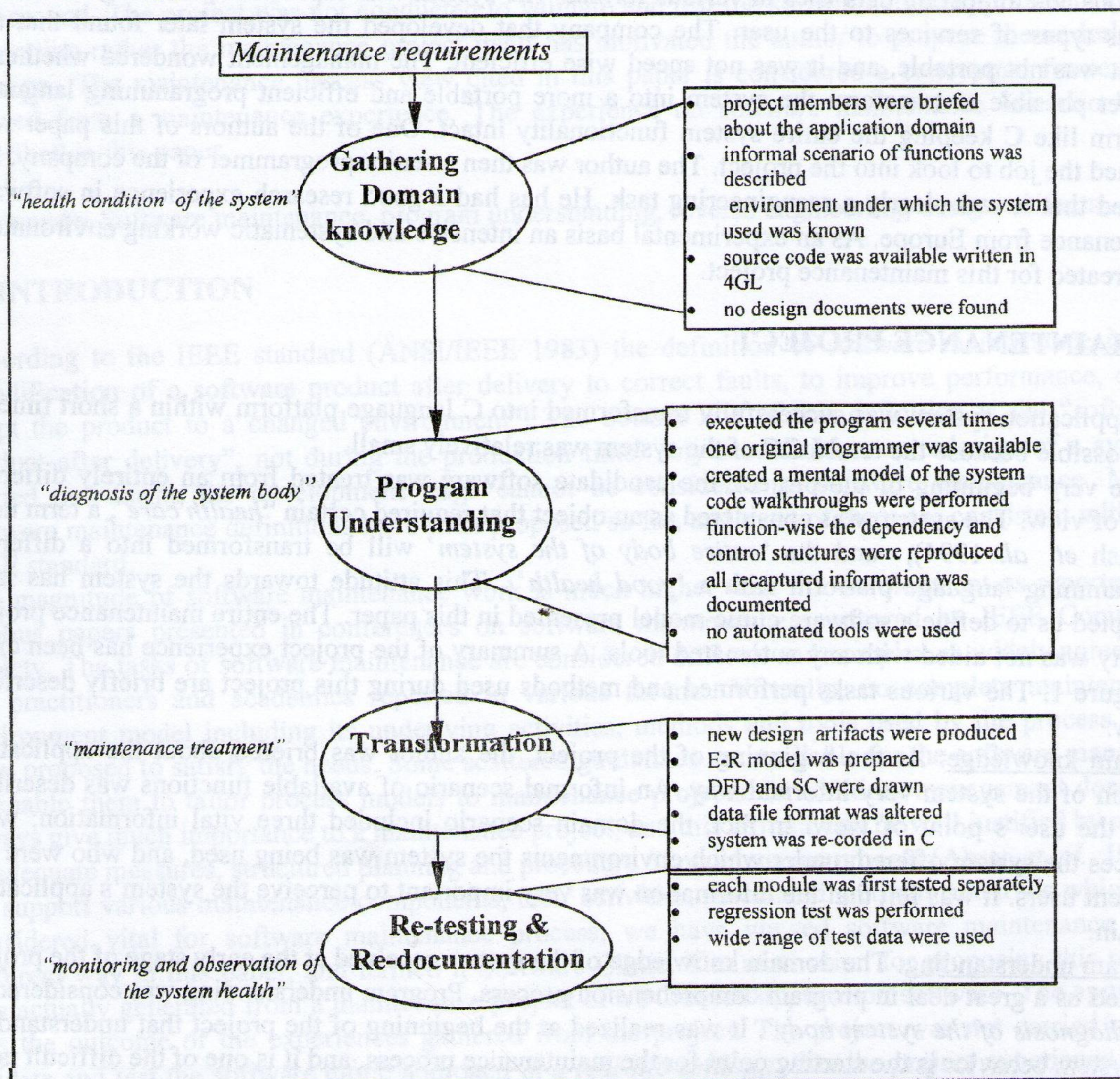

Figure 1: Summary of the Software Maintenance Project Experiences

Transformation: This was the '*treatment*' part of the maintenance project. The system was redesigned based on the information obtained, and some additional new design characteristics were also included to increase the performance of the system. Entity relationship model was created to keep track of relationships among different data objects. Data-flow diagrams and structured charts were drawn to show the functionality and architecture. The design rationale of the new version was slightly different than the older version. The call structures, control flows, roles of variables were changed to eliminate the design flaws found in the old system. All these documents were verified and inspected by the project members several times. The system was then coded in C programming language. The programming task was rather painless and smooth compared to program understanding process.

Re-testing: We later termed this task as '*observation of the system body*' with test data. The program code was tested with varieties of test data. Each module was first tested separately, then regression test was perform to check the completeness of the system. The test data was generated from a wide range of possibilities. It included not only the real data of the application domain, but also unrealistic data was fed into the system to determine its behavior. The speed and overall performance of the new system was much higher than the earlier version. Total disk space taken by the entire system was much less than the older one.

4 LESSONS LEARNED

It has also been realized that it is easy to recover the complex artifact for the project members if they are familiar with the implementation domain of the system and the size of the software is small. The scale of the candidate software is an important aspect in software maintenance (Bennett, 1995). Based on this maintenance project experience we have later developed a maintenance model: Software Clinic. We are now in position to describe the basic components of our model and their associated methods, tools, output products of the components and sources of input materials of the components.

5 COMPONENTS OF SOFTWARE CLINIC

It is believed that software maintenance process should consist of an orderly and well-defined set of activities to achieve a certain goal. The major activities of the process must be able to create a well-defined maintenance environment. These activities must be aided with predefined methods, established tools, and the sources of input materials for the activities. Based on these ideas, we view software maintenance process completely from a different perspective. We see the process from a point of view that much more resembles with the medical clinic management approach. The maintenance process can easily be considered as the '*health care of the system*' as termed in (Ash *et. al.* 1994). We have established an analogy between software maintenance and health care system in our proposed model. This clinic approach has four major components:

- Informal Scenario and Consultation
- Formal Diagnosis
- Maintenance Treatment, and
- Monitoring and Observation.

Each of these components is associated with some methods, tools, its intermediate output products and sources of input materials to be fed to other subsequent activities. There is a feedback loop exists between two subsequent components. The software clinic view of the maintenance process is cited in figure 2. Its major components are briefly described as follows.

5.1 Informal Scenario and Consultation

Objective: *"Perceive the nature of system health and body"*

The activities of this component are activated whenever the process receives a maintenance request. The user expresses the maintenance requirements and the behavior of the software system in a scenario form. This scenario includes maintenance requirement specifications, domain knowledge of the system, and the functional behavior of the software. This scenario assists maintenance programmers to construct a mental model of the system. To capture some of the requisite knowledge needed to support maintenance work, this information is essential. This component has five major activities.

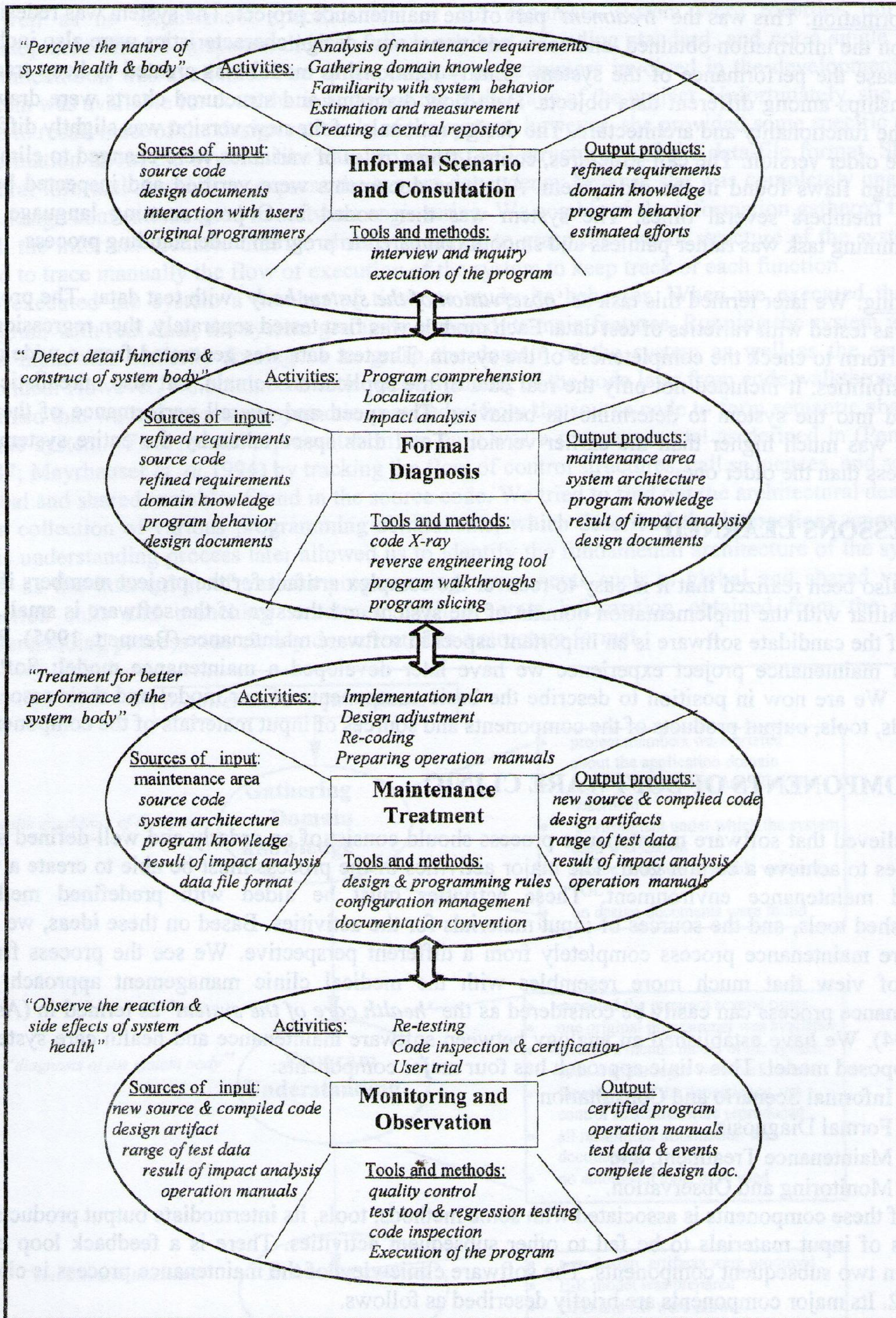

Figure 2: Software Clinic: A View of Software Maintenance Process

Activities:

Analysis of maintenance requirements: The maintenance requirement acts as a navigator of the process, it guides the maintenance process in which direction the project should ahead, which activities be given priority, and the magnitude of the maintenance work load. Software community has already accepted that the requirements of software maintenance are quite different than the software development. Even the approach and the view of these two processes differ from each other

significantly (Skramstad *et al.*, 1992; Khan *et al.*, 1996). It is also important to examine whether the new change requirements will benefit the user of the system or not. How the new requirements will justify the effort of the project in terms of cost, technology and performance. The repeating and overlapping requirements should also be filtered out at this level. This activity will decide whether the project should further proceed or not.

Gathering domain knowledge: In this activity, an informal scenario of the candidate software system is obtained including the domain of the system. Gathering information on the candidate system domain and application domain is the first step in software maintenance. A*pplication domain* includes the application area on which the system was built i.e., administration, banking, space shuttle; and *system domain* includes the programming languages used, supporting operating systems, hardware configuration etc. The domain knowledge is abstracted informally from the discussions with users and original developers if available, and by executing the system with real data. The project members use this information to build a primary mental model of the system. It gives a sense of the nature and familiarity to the system.

Familiarity with system behavior: The behavior could be expressed in terms of some atomic events. This activity endeavors to collect and represent a body of relevant knowledge, which determines the behavior of the system. This information is needed to populate the central repository with descriptions of a set of basic modules including its behavior and input-output patterns. The central repository can then be augmented with additional module descriptions derived in the next component.

Estimation and measurement: A structured report must be prepared to estimate the possible cost and benefit, and the viability of the project. Resources must be estimated, and schedule is to be chalked out. Priority of the various tasks must be defined, and this priority list can be shifted around any time as required.

Creating a central repository: All information gathered in this phase is stored together with their attributes and relationships in a central database. The database cannot be updated, but can be only populated with data. If it is found that some information collected earlier was not entirely correct, this couldn't be wiped out completely from the repository. Every piece of system information and every act performed will be recorded during the entire project duration.

Tools and methods:

Interview, inquiry, and execution of program with real data.

Sources of input materials:

Source code, design documents if available, interactions with the users, consultation with the original developer of the system.

Output products of the component:

Central repository, estimated efforts and measurements, application domain knowledge, program behavior, refined maintenance requirements.

5.2 Formal diagnosis

Objective: *"Detect detail functions and construct of the system health and body"*

There already exists mature technology in this area such as debuggers, reverse engineering tools, program slicing, impact analysis and so on. Information on such tools and techniques are found (Carmichael *et al.* 1995; Arnold, 1993). It has three major activities as outlined below.

Activities:

Program comprehension: The source code must be x-rayed, as we term them, if the design documents are missing or unreliable. Code X-ray is a synonym of reverse engineering technique first referred in (Khan *et al.* 1996)

A major part of the total efforts spent on a software maintenance project is consumed in program understanding task (Chapin, 1988). It was obvious that maintenance programmers have to capture the intentional structure of a software system and its embedding organization in terms of dependency relationships among programming components. The components are dependent on each other for some goals to be achieved, functions to be carried out and resources to be exchanged. Some automatic

and semi-automatic program understanding tools are available in the market.

Localization: The exact locations in the source code where the proposed maintenance to be performed are identified. It is also important to trace other components, which are directly or indirectly related to the maintenance locations in the system.

Impact analysis: Impact analysis is an important and well-known problem in software maintenance. A good number of research works has been published on this area in (Bohner *et al.* 1996). A high level definition of impact analysis has been proposed in (Queille *et al.* 1994) as "The task of assessing the effects of making a set of changes to a software system".

The impact analysis is triggered to trace expected and unexpected behavior of the software system when a set of changes is proposed to the system. Although, there is no widely accepted rule of when impact analysis should take place in the maintenance process, but it has been suggested in (Barros *et al.* 1995) that this analysis is to be performed before the actual modifications are implemented in the source code.

Impact analysis can be classified into two categories: *direct* and *indirect*. In direct impact, when a modification is introduced it effects directly to the adjacent components, which are visibly related to the change site. The indirect impact is considered most dangerous and difficult to trace. When a component behaves unexpectedly due to some changes in the other components not directly related to that component is considered indirect effect. The analysis is applied to identify potential side effects at the source code level, but also to determine the impacts at the design level, user documents etc (Barros *et al.* 1995). Any change itself can be categorized as a cause of three types of ripple effect: potential, obvious and unknown. Failure to detect any of these ripple effects in safety critical or financial systems will lead to a catastrophic effect as a whole. Therefore, it is important to control the process more rigorously to trace all types of ripple effect.

Tools and Methods:

Code X-ray or Reverse engineering tool, program walkthroughs, program slicing.

Source of input materials:

Maintenance requirements, source code, central repository, original programmer, documents produced in the previous component.

Output products of the component:

System architecture, program domain, names of the modules where the modifications to be introduced, result of the impact analysis, complete recovered design documents.

5.3 Maintenance Treatment

Objective: *"Treatment for better performance of the system health and body"*

In this component the planned modification is implemented at the code level of the system. The design documents as well as user manuals are rewritten accordingly. The component has four distinct activities: implementation plan, design adjustment, re-coding, and preparing operation manuals.

Activities:

Implementation Plan: A detail plan is to be worked out on how the modification is to be introduced to the software. This activity is based on the information received from the earlier components.

Design adjustment: The new design is produced to accommodate the maintenance requirements as well as the old design documents are to be updated.

Re-coding: The programming will be completed according to the updated new design decision. The re-coded site in the source code is configured with the entire system.

Preparing operation manuals: According to the new version of the software, the user manuals are to be prepared. The user interface should be clearly described.

Tools and methods:

Design and programming rules, configuration management, compilation, documentation convention.

Sources of input materials:

System architecture, program knowledge, names of the modules where the modifications to be introduced, data file formats, result of the impact analysis.

Output products of the component:

Source code of the new system version, executable code, design artifacts, design artifacts of the previous version, operation manuals, report on impact analysis, and the possible range of the test data.

5.4 Monitoring and Observation

Objective: *"Observe the reaction and side effects of the system health"*

This component comprises three activities as follows:

Activities:

Re-testing: Regression testing has to be performed with a wide range of possible test data, capturing the performance of the modified system with the previous version (Nyary *et al.* 1995). Every path and control branches particularly the modified area of the system must pass a rigorous test session with real data input. Weinberg (Weinberg, 1983) reported in his list of large software disasters that the top three systems in the list are one-line changes that were not tested. It is, therefore, important to perform all possible tests before it is certified.

Code inspection and certification: The new version of the system must be inspected whether the prescribed programming rules are followed, design documents are consistent with the code, and test data are adequate and accurate.

User trial: The system will finally be installed in the user's work premises. The users with their business data will execute it under the intended-working environment. This can be performed in the user site, but must be closely monitored by the project team members for a certain period.

Tools and methods:

Test tools, quality control, regression testing, and execution of the entire program.

Sources of input materials:

Source code of the new system version, executable code, user manual, design artifacts, design artifacts of the previous version, report on impact analysis, and the possible range of the test data.

Output products of the component:

Certified program with source code, users manual, design documents, test data and events.

In our framework, *program comprehension, software change impact analysis*, and *re-testing* are considered most critical and important activities. However, if the updated and reliable design documents are available during the maintenance project then program comprehension becomes simple.

6 RELATED WORK

Researchers and practitioners have proposed a number of software maintenance processes over the years. They either tried to explain the actual process of software maintenance more precisely or prescribed a better and effective process model. Some of the specific works on software maintenance model are found particularly in (Skramstad *et al.* 1992; Chernika *et al.* 1994; Ino, 1992; Harjani *et al.* 1992; Desclaux *et. al.*, 1992; Hinley *et al.* 1992). The fundamental difference of software clinic model and other maintenance environment is the view and attitude towards the process. Most of the models proposed a list of tasks or procedures and their order without addressing the underlying methods, tools, input and output information associated with each task or procedure.

7 FURTHER WORK

The software clinic view presented in this paper can be developed further using more formal process modeling approach to get automated tool support. To enrich the technical contents of the model, more formal relationships among the task components can be established. The model can be justified with its application to a project.

This approach may be integrated into the entire software life cycle which may examine its ability to synchronize with the development process as well. In such a model, it is important to show that the development environment supports the underlying maintenance method and activities within a fuller software life cycle framework. Because, there is a need for a software maintenance process for the practitioners supporting a fuller software life cycle (Foster, 1992; Chapin, 1988; Rombach *et al.* 1992).

8 CONCLUSION

This paper focuses on another way of viewing the software maintenance process environment. It is presented to specify the maintenance process more visibly. This approach has visualized the type and distribution of activities needed for software maintenance process. We define an approach as a set of tasks coupled with the responsibilities to individual group and process components. This approach is believed to remove the overlap of tasks and gaps between the activities while it provides coordination, thereby increases the overall effectiveness of the maintenance process.

REFERENCES

Abran, A., Maya, M. (1995). 'A sizing Measure for Adaptive Maintenance Work Products', *IEEE Proceedings Conference on Software Maintenance*, 1995, 286- 294.

ANSI/ IEEE (1983). ANSI/IEEE Standard 729-1983.

Arnold, R. S. (1993). *Software Reengineering, IEEE Computer Society Press*, Los Alamitos, 1993.

Ash, D., Alderete, J., Yao, L., Oman, P., Lowther, B. (1994). 'Using Software Maintainability Models to Track Code Health', *IEEE Proceedings Conference on software Maintenance*, 1994, 154-160

Barros, S., Bodhuin, T., Escudie, A., Queille, J. P., Voidrot, J. F. (1995). 'Supporting Impact Analysis: a Semi-Automated Technique and Associated Tool" *IEEE Proceedings Conference on Software Maintenance*, 1995, 42- 51.

Bennett, K. (1995). 'Legacy Systems: Coping with Success', IEEE Software, January 1995, 19-23.

Bohner, S. A., Arnold, R. S. (1996). *Software Change Impact Analysis*, IEEE Computer Society press, Los Alamitos, 1996.

Carmichael, I., Tzerpos, V., Holt, R.C. (1995). 'Design Maintenance : Unexpected Architectural Interactions', *IEEE Proceedings Conference on Software Maintenance*, 1995, 134- 137

Chapin, N. (1988). 'Software Maintenance Life Cycle', *IEEE proc. conf. on Software Maintenance*, 1988, 6-13.

Chernika, R., Overstreet, C. M., Cadwell, A., Ricci, J. (1994). 'Issues in Software Process Automation – From a Practical Perspective', *IEEE Proceedings Conference on Software Maintenance, 1994*, 109-118.

Desclaux, C., Ribault, M. (1991). 'MACS: Maintenance Assistance Capability for Software a K. A. D. M. E', *IEEE Proceedings Conference on Software Maintenance*, 1991, 2-12.

Foster, J. (1992). 'Survey Report', *European SIG in Software Maintenance Newsletter* Issue 3: June 1992. 5-7.

Harjani, D. R., Queille, J. P. (1992). 'A Process Model for the Maintenance of Large Space Systems Software', *IEEE Proceedings Conference on Software Maintenance*, 1992, 127-136.

Hinley, D., Benneth, K. (1992). 'Developing a model to manage the software maintenance process', *IEEE Proceedings Conference on Software Maintenance*, 1992, 174-182.

Ino, M. (1992). 'Current State of Software Maintenance in Japan: In Depth View', *IEEE Proceeding Conference on Software Maintenance*, 1992, 27-29.

Khan, M. K., Rashid, M. A., Lo, B. W. N. (1996). 'A Task-Oriented Software Maintenance Model', *Malaysian Journal of Computer Science*, University of Malaya, Vol.9, No. 2, Dec., 1996. 36-42.

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

Nyary, E., Sneed, H.M. (1995). ' Software Maintenance Off loading at the Union Bank of Switzerland', *IEEE Proceedings Conference on Software Maintenance*, 1995, 102- 108.

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

Queille, J. P., Voidrot, J.F., Wilde, N., Munro, M. (1994). 'The Impact Analysis Task in Software Maintenance: A Model and a Case Study', *IEEE Proceedings Conference on Software Maintenance*, 1994, 234-242.

Rombach, H. D. and Basili, V. (1988): 1A Panel Discussion, Position Statement, *IEEE Proc. Conf. on Software Maintenance*, 1988.

Skramstad, T., Khan, M. K. (1992). 'A Redefined Software Life Cycle Model for Improved Maintenance', *IEEE Proceedings Conference on Software Maintenance*, 1992, 193-197.

Weinberg, G. (1983). 'Kill that Code!', *Infosystems*, , August 1983, 48-49.