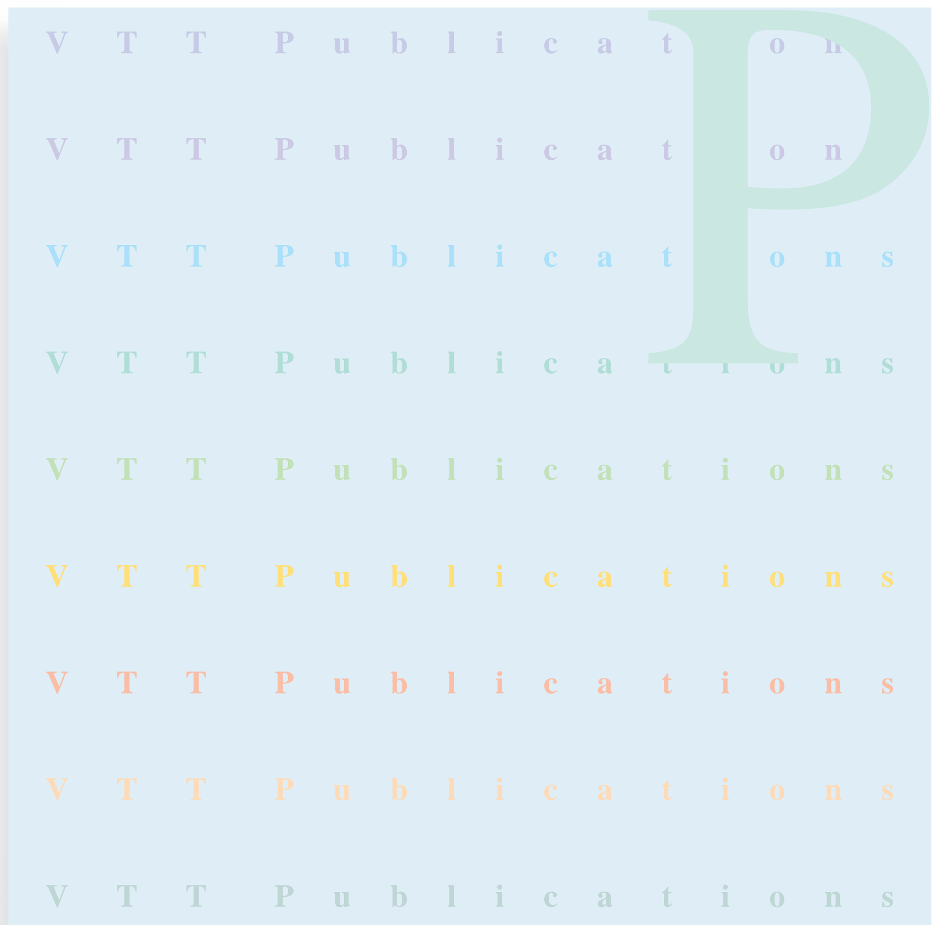
416 VTT PUBLICATIONS

Minna Mäkäräinen

Software change management processes in the development of embedded software



TECHNICAL RESEARCH CENTRE OF FINLAND ESPOO 2000

VTT PUBLICATIONS 416

Software change management processes in the development of embedded software

Minna Mäkäräinen

VTT Electronics

Academic Dissertation to be presented with the assent of the Faculty of Science, University of Oulu, for public discussion in the Auditorium L10, Linnanmaa, on August 26th, 2000, at 12 noon.

TECHNICAL RESEARCH CENTRE OF FINLAND
ESPOO 2000

ISBN 951–38–5573–2 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–5574–0 (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)

Copyright © Valtion teknillinen tutkimuskeskus (VTT) 2000

JULKAISIJA – UTGIVARE – PUBLISHER

Valtion teknillinen tutkimuskeskus (VTT), Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

Statens tekniska forskningscentral (VTT), Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

Technical Research Centre of Finland (VTT), Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Elektroniikka, Sulautetut ohjelmistot, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde (08) 551 2111, faksi (08) 551 2320

VTT Elektronik, Inbyggd programvara, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel (08) 551 2111, fax (08) 551 2320

VTT Electronics, Embedded Software, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland
phone internat. + 358 8 551 2111, fax + 358 8 551 2320

Otamedia Oy, Espoo 2000

Mäkäräinen, Minna. Software change management processes in the development of embedded software. Espoo 2000, Technical Research Centre of Finland, VTT Publications 416. 185 p.+ app. 56 p.

Keywords software change management, software configuration, software maintenance, process improvement, process modelling, process analysis

Abstract

The goal of the research presented in this thesis is to examine software change management processes in order to identify essential change management problems and improvement requirements, to define processes which would aid in solving these problems, and give an example of how these processes can be implemented in practice.

The subjects of the empirical research part of the study have been four Finnish companies which develop embedded software. Therefore, the focus of the study is on the processes which are used in developing embedded software. However, the literature study explores the problems of software change management from a more generic viewpoint, and can also be used as a reference for software development done for other purposes.

Three of the four case studies are used in deriving the generic change management problem classes and process descriptions. These three case studies include only analysis of the change processes and problems related to them. The fourth case study is used for illustrating how the proposed problem classification can be used in change process analysis, and how the proposed process models can be instantiated in practice. The change processes were not only analysed, but the study also included the definition of new processes, the planning of their implementation, and the implementation and enactment of the new processes in the organisation.

Preface

This thesis summarises the results of the research I was involved in while working at VTT Electronics. It explores the problems of software change management on the basis of a literature study and practical case examples. I started to contemplate the process of software change management when I was working in the ESPRIT project called AMES. Later, I had the opportunity to apply the solutions and ideas developed in the AMES project in projects for Finnish companies. The literature study part of the thesis was mainly done while I was working as a visiting researcher at Fraunhofer IESE, Germany. I finalised the writing of the thesis while working at Nokia Mobile Phones. I want to thank Dr. Veikko Seppänen and Dr. Jorma Taramaa for encouraging me to continue my studies at VTT Electronics, Dr. Dieter Rombach for the opportunity to work at Fraunhofer IESE, and Dr. Pekka Isomursu and Dr. Pertti Huuskonen for helping me through the last phases of the writing process at Nokia Mobile Phones. Unfortunately, I am not allowed to name the helpful persons from the companies I studied during the research process. Anonymous thanks to all of you!

This work has been financially supported by my employers. I also received financial support, which is gratefully acknowledged, from the following foundations: Seppo Säynäjäkankaan rahasto, Tauno Tönningin säätiö, and the Finnish Cultural Foundation.

Dr. Ilkka Tervonen has been the advisor of my post-graduate studies for seven years. I cannot thank him enough for all the work he has done for my studies. The reviewers of the thesis, Dr. Cornelia Boldyreff and Dr. Karlheinz Kautz, gave me valuable and profound comments, which dramatically improved the quality of the thesis. I am very grateful for the time and effort they used for reviewing my work.

Finally, I want to thank my family and friends for their love and support. My parents have given me an enormous amount of support and understanding, still giving me the space to live my own life. And Pekka, my soulmate, thank you for all you are!

Oulu, June 2000 Minna Mäkäräinen

Contents

Abstract 3

Preface...4

List of symbols 8

1 Introduction 9
   1.1 Research questions and overview of research methods......................10
   1.2 Scope of the research 11
   1.3 Structure of the thesis 12

2 Research method 16
   2.1 Overview...16
   2.2 Research approach 16
   2.3 Research methods used in the case studies in the problem analysis stage 16
   2.4 Research methods used in the construction stage...............................23
   2.5 Research methods used in the demonstration stage............................24
      2.5.1 First phase: Analysis of current practices.....................................26
      2.5.2 Second phase: Definition of goals for new practices 29
   2.6 Evaluation of the results 30
   2.7 Research process...31
   2.8 Summary 33

3 Analysis of related work 34
   3.1 Overview...34
   3.2 Scope of software change management 34
   3.3 Why is software change management difficult?...................................41
   3.4 Software change management – Product dimension 44
   3.5 Software change management – Process dimension 46
      3.5.1 Olsen's change management model 48
      3.5.2 V-like change management model...50
      3.5.3 Ince's change process model 52
      3.5.4 The AMES model 55
      3.5.5 Spiral-like change management process..56
   3.6 Software change management – Technological dimension.................58
   3.7 Summary 62

4 Analysis of the state of the practice 63
   4.1 Overview...63
   4.2 Summary of case studies 63
      4.2.1 Case one 64
      4.2.2 Case two..70
      4.2.3 Case three 74
   4.3 Summary 78

5 Software change management problems....81
5.1 Overview....81
5.2 Classification of problems and improvement requirements in change management processes....81
5.2.1 Effectiveness problems....82
5.2.2 Communication problems....84
5.2.3 Analysis and location problems....87
5.2.4 Traceability problems....89
5.2.5 Decision-making problems....90
5.2.6 Tool-related problems....91
5.3 Summary....96

6 Generic change management process model....97
6.1 Overview....97
6.2 Background....97
6.3 Layered change processes....98
6.3.1 Product-level changes....102
6.3.2 Project-level changes....103
6.4 Generic change management process....104
6.4.1 Trivial defect correction....106
6.4.2 Defect correction....109
6.4.3 Requirement-level modification....111
6.4.4 Improvement proposal....111
6.5 Relation of process levels and process types....112
6.6 Comparison to other models....112
6.7 Summary....114

7 Implementation....116
7.1 Overview....116
7.2 Operational organisation....116
7.3 Process management....116
7.3.1 Reviews of change requests....117
7.3.2 Monitoring the change requests....117
7.4 Quality responsibilities related to change requests....118
7.4.1 Technical review....118
7.4.2 Testing....118
7.4.3 Other change types....119
7.5 Sources of change requests....119
7.6 Description of the change management process....122
7.7 Existing change management support....125
7.8 Problems and improvement proposals related to change management....127
7.8.1 Effectiveness problems....127
7.8.2 Communication problems....127
7.8.3 Analysis and location problems....128
7.8.4 Traceability problems....130
7.8.5 Decision-making problems....131

7.8.6 Tool-related problems...131
7.9 Description of the implementation solution 132
7.9.1 Vocabulary...132
7.9.2 Instantiation of the generic processes in the case study 133
7.9.3 Selection of tool environment..153
7.10 Implementation and deployment of the defined solution.................155
7.11 Summary 159
8 Evaluation of results 160
8.1 Introduction...160
8.2 Evaluation of the change management problem classification............160
8.3 Evaluation of the change management process model 162
8.4 Experiences from the implementation and deployment 167
8.5 Summary 171
9 Conclusions..172
9.1 Answers to research questions..172
9.2 Generalisation of the results 173
9.3 Future research..174
10 Epilogue 176
References...177

Appendix A: Case one – aerospace organisation

Appendix B: Case two – consumer electronics organisation

Appendix C: Case three – telecommunication organisation

List of symbols

AMES An ESPRIT project on tools and methods for application management

CM Change Management

CMM Capability Maturity Model developed by the SEI

CMS Code Management System, version control tool by Digital

CS Cellular System

ESPRIT European Strategic Program for Research and Development in Information Technology

GQM Goal-Question-Metric method

HOOD Hierarchical Object Oriented Design, an object oriented design language

HW Hardware

OS Operating System

Pr^{2}imer Practical Process Improvement for Embedded Real-Time Software, a process improvement service package by VTT Electronics

SADT Structured analysis and design technique

SCM Software Configuration Management

SEI Software Engineering Institute at the Carnegie Mellon University, Pittsburgh, USA

SW Software

UI User interface

VTT Technical Research Centre of Finland (Valtion teknillinen tutkimuskeskus, in Finnish)

1 Introduction

The problem of managing software changes has gained a lot of attention in recent years. The European Space Agency's Ariane 5 Flight 501 rocket disaster was caused by poor change management in reused software parts. The year 2000 problem has required a great deal of effort on updating and verifying the operationality of software systems all over the world. The combination of year 2000 modifications and the inception of the European Monetary Union between 1998 and 2002 will result in fundamental modifications to most legacy systems used today; to name but a few of the recent change management crises.

At the same time, the importance of software development in electronics has increased rapidly during the past ten to fifteen years. One of the basic reasons for this is that new types of software-intensive products have emerged, especially in communications and consumer electronics. This growth has put pressure on companies and led to the adoption of effective software engineering processes.

During the early days of embedded systems development it was normal to develop the different technological parts of the system, such as hardware, system software and application software, separately. At present, the development of different product technologies must be done concurrently. Time-to-market constraints require better integration of the different product technologies. During the development of the hardware and software components, the system requirements often change and evolve. The changes relate to all parts of the product, creating problems and special requirements for change management.

Customer-specific features of electronics products are often implemented by means of application software rather than hardware. The existence of several customer-specific versions of the same application software creates problems in managing the modifications. The applications may share most parts of the source code, or they may be totally separate with redundant source code parts. Both situations are problematic. If the applications share parts of the source code, changes made in one of the applications are reflected in the other applications through the common parts and may create erroneous situations or unexpected behaviour. When the applications are totally separate, a modification

implemented in one application has to be repeated several times if it is relevant to some other applications.

The use of reusable software components has increased in the development of embedded software. The software is not programmed from scratch, but rather assembled using reusable software components. This kind of a development process sets special requirements for change management, such as the maintenance of a component library, predicting the impacts of changes in reused components, analysing the behaviour of the combination of reused components and new software, managing versions of reused components, etc.

In the case of embedded software, needs for changes are not only created by the application software, but also by the hardware, the system software and other parts of the product (Taramaa 1998). If the hardware components of the product have to be modified for example because of a design error, a corresponding modification may also be necessary in the application software. The hardware and software components are usually designed concurrently. Since the hardware environment is not stable when the software requirements are defined, redefinition of both application and system software is often unavoidable. Therefore, continuous interaction between application software, hardware and system software developers and maintainers is required.

1.1 Research questions and overview of research methods

The goal of the thesis is to find answers to the following research questions:

- What are the essential change management problems and improvement requirements found in developing embedded software?
- What kind of a management processes would help in responding to these requirements?
- How can the proposed processes be implemented and enacted in practice?

The first research question is studied using a literature study and by analysing the change management processes in three organisations. The case study organisations are analysed using semi-structured focused interviews complemented with tool and document analysis and analysis of process and product related measurement data, when available. Using these sources, essential problems in change management are identified and analysed, and improvement requirements for change management practices are proposed. As a result of the analysis of change management problems a characterisation of the problem domain is derived, and a classification of the typical problems in software change management is developed. The problems are related to the special features of companies developing embedded software.

The second research question examines what kind of generic change management process types would provide support for responding to the software change management problems identified. As a result, a generic model of software change management processes is derived.

The proposed generic process model is applied in one case study in order to implement new change management processes. The implementation process is described in order to answer to research question three. The implementation solution will be described, including (1) a description of the initial change management status and the requirements for improving software change management practices in the organisation, (2) an analysis of change management problems using the problem classification proposed, (3) the process models instantiated from the generic change management processes presented, and (4) a description of the technical implementation. Finally, the experiences of using the new change management solutions based on the proposed model are evaluated.

The used research methods are described in detail in chapter 2.

1.2 Scope of the research

The scope of software change management is restricted in this thesis to changes in the software itself, i.e. source code, design documentation, test cases, user manuals, etc. It is not defined to cover issues related to changes in the development environment, human resources, project schedules, etc.

The industrial case studies presented in the thesis are all from the domain of embedded software. Therefore, the thesis proposes a change management problem classification and process model for organisations developing embedded software. Since other application domains have been studied only from the viewpoint of the research presented in the literature, a thorough analysis of differences between the domain of embedded software and other domains was not possible. The specific requirements for the development of embedded software are analysed by mapping the identified problem areas and improvement requirements onto the special features of the embedded software.

The thesis focuses on the change management process. The tools and technological solutions for specific process activities, such as program comprehension, impact analysis or regression testing, are not covered in detail.

1.3 Structure of the thesis

The structure of the thesis is illustrated in Figure 1. Typical problems and improvement requirements for software change management are derived from the literature study and the three background case studies. The goal of combining the literature study and case studies is to provide both state-of-the-art and state-of-the-practice viewpoints for analysing the problems and defining a change management process model. The presented process model is then applied in implementing new change management solutions in one case organisation. The implementation includes evaluation of the initial status of change management in the organisation by utilising the problem classification defined, tailoring of the generic change management processes for the needs and requirements of the case organisation, and implementation of the new change management solution.

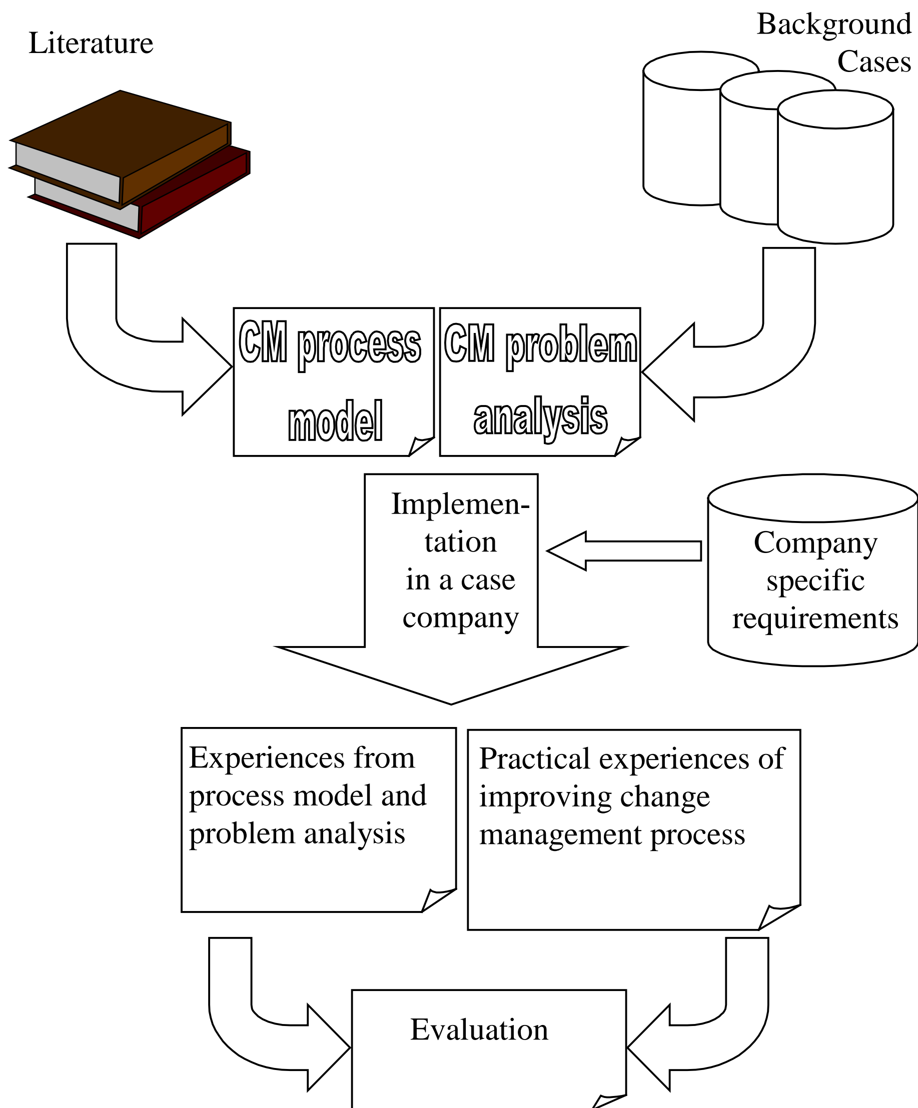


Figure 1. Structure of the thesis.

Chapter 1 of the thesis presents the research problems and the scope of the thesis. It also presents an overview of the structure of the thesis and gives a brief summary of the main chapters.

Chapter 2 explains the research methods used in different phases of the construction of the thesis.

Chapter 3 summarises published research work related to the subject of the thesis.

Chapter 4 examines the change management processes and problems in three companies developing embedded software. The chapter summarises the case studies. More detailed case material is included in appendices A to C. The chapter aims at providing a practical viewpoint to defining the typical change management problems presented in chapter 5 and the change management processes presented in the chapter 6. Chapters four and five describe how this thesis defines software change management.

Chapter 5 presents a classification of change management problems and improvement requirements derived from the literature study and the background case studies summarised in the previous chapter. The problems and improvement requirements are presented in the form of a problem classification framework. Where possible, the specific problems related to the special features of embedded software development are presented separately for each problem class.

Based on the problems and requirements defined, chapter 6 proposes a generic change management process model, which aims at providing a general framework for building a process environment which would respond to the problems and improvement requirements of the organisation. This chapter describes how change management processes are defined in this thesis.

Chapter 7 applies the proposed problem classification by analysing the change management requirements in one case organisation. The initial status of change management is briefly described, and the identified problems and improvement requirements are presented using the classification framework presented in chapter 5. To tackle the problems encountered, a new change management process environment is designed by applying the process model proposed in chapter 6. The instantiation of the generic process model to the special needs of the case organisation is presented. Finally, the implementation of the instantiated model is presented.

Chapter 8 evaluates the experiences gained and results obtained from using the proposed problem classification and process model for implementing a new

change management solution in the case organisation. Also, guidelines for the successful deployment of the new solution are given. The defined solution had been in active use in the organisation for some years at the time of writing this thesis.

The last chapter concludes the research work presented in the thesis, gives a summary and analysis of the answers obtained to the original research questions and speculates on future research on the topic.

2 Research method

2.1 Overview

This chapter describes the research approach adopted in constructing this thesis, and the research methods used in different research stages.

2.2 Research approach

The research approach selected for this work is mainly constructive. Three main research stages can be identified: (1) Problem analysis, (2) Construction and (3) Demonstration. The problem analysis examines the domain of software change management using a literature study and three case studies. Based on the problems and improvement requirements identified in the problem analysis stage, a classification framework for typical problems in change management, related especially to the specific features of embedded software, is derived. Furthermore, the results of the problem analysis are used in the construction stage to derive a generic change management process model. The last stage instantiates and demonstrates the proposed solutions in one industrial case study. The problem classification is used in defining the case requirements for an improved change management solution, and the generic process model is used for deriving the process instantiations for one organisation.

2.3 Research methods used in the case studies in the problem analysis stage

The problem of software change management was analysed using a literature study and three case studies. The first two case studies (narratives are presented in Appendices A and B) were conducted using qualitative, focused interviews. The conceptual schema of the interviews is presented in Figure 2. The conceptual schema was used in deriving an interview framework, which included the focus areas that should be addressed and discussed during the interview sessions. The interview framework used in the first two case studies is described in detail in (Mäkäräinen 1996). The framework used in the third case study is dif-

ferent in structure and wording, but the conceptual schema used in deriving it is the same. The interview framework was also used in writing notes on individual interview sessions. The conceptual schema for the interviews was the same in all three case studies. However, the interview framework was constructed separately for each case study. The experiences from previous case studies were used in constructing a new interview framework. Also, the individual needs and special features of the studied case organisation affected the interview framework. The same framework was then further refined into a framework of the in-case summary narrative. Since the interview frameworks for each case study were slightly different, the resulting case narratives also differ in format, as can be seen from Table 1.

Table 1. Structure of case narratives.

Case one	**Case two**	**Case three**
Operational organisation	Operational organisation	Introduction, Context of the analysis
Description of application domain a) Technical description b) Documents	Description of application domain a) Technical description b) Documentation	Description of the product
Application development a) General information b) Development processes c) Development methods/techniques d) Development environment e) Development documentation	Application development a) Development processes b) Development methods/techniques c) Development environment d) Development documentation	Organisation of change management a) Two layers of change processes b) Documentation c) Quality management of change management processes d) Human resources management

Table 1. Continues.

Case one	**Case two**	**Case three**
		Change requests a) Defects found in testing b) Defects found in reviews c) Added and modified features d) Improvement propos-als e) Sources of change requests
Current application management practices a) General information b) Application man-agement organisation c) Application manage-ment process d) Application man-agement environment	Current application management practices a) Application manage-ment organisation b) Application man-agement process c) Application manage-ment meth-ods/techniques d) Application man-agement environment e) Maintenance history	Current change proc-esses a) Defects found in testing b) New/modified fea-tures c) Improvement propos-als d) Errors found in re-views
		Change management methods and techniques

Table 1. Continues.

Case one	Case two	Case three
Main problems in application management	Main problem in application management	Problems in change management
Application management requirements	Requirements for improving application management a) Process support b) Configuration and version management c) Reverse engineering d) Modification request management e) Regression testing f) Impact analysis	

Focused interviews were also used as a main information source in the third case study, but the information extracted by means of interviews was complemented with quantitative data, i.e. data acquired from the GQM-based measurement programme (van Solingen & Berghout 1999), and change management related document and tool analysis. The third case company had a running measurement programme which provided data related to changes. Triangulation between results of multiple data collection methods was used to achieve stronger substantiation of results and to increase the confidence in the findings. The methods also supported each other; the document and tool analysis performed before the interviews familiarised the interviewees with the concepts used in the company and decreased the need to explain standard procedures and tools in detail during the interview sessions.

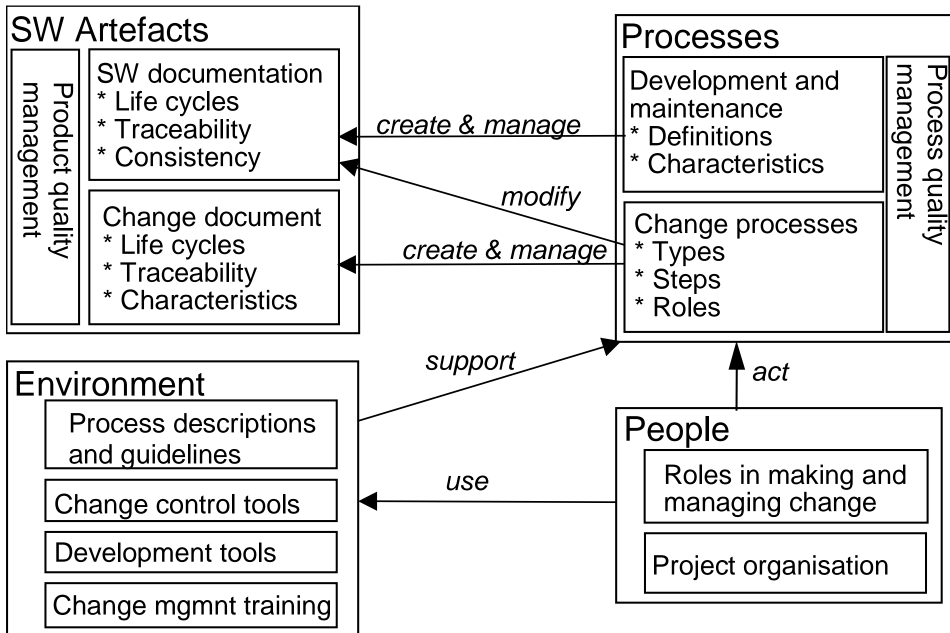


Figure 2. Conceptual schema.

The first two case studies were done in an ESPRIT project called AMES (AMES 1993). The project aimed at improving the application management process as one business process of a company by providing a methodological framework for application management, a set of tools for supporting the most critical activities of application management, and an integrative infrastructure to ensure the interoperability of the tools provided by the project and commercial off-the-shelf tools. The project defined application management as:

Contractual responsibility for the management and execution of all activities related to the maintenance and evolution of existing applications.

The first two case studies aimed not only at learning about the change processes and related problems in the case study organisations, but also at creating requirements for the change management methods and tools to be developed in the AMES project. Therefore, a list of requirements was generated by the representatives of the case study organisations. This list was not generated in the third case study, since it was not part of the AMES project, and did not aim at

implementing tools or methods for change management. Instead, this study focused on identifying change management problems.

Since one of the main goals of the case studies was to reveal problems and bottlenecks in the change management, the interviewing technique was considered a suitable method for performing the case studies. The strength of the interview method is its privileged access to the common understanding of the subjects, the understanding that provides their worldview and the basis for their actions (Kvale 1996). The personal perspectives of the subjects and the interviewer provided a distinctive and sensitive understanding of the problems faced in the complex phenomena of change management. Interviews have been evaluated to be effective in exploring what problems the subjects see as most important, how the subjects place themselves in various classification schemes, etc. (Goguen & Linde 1997).

The interviewees were selected using theoretical sampling, i.e. they were chosen for theoretical, not statistical, reasons. The goal of theoretical sampling is to allow replication and extension of the emergent results by examining extreme situations and polar types (Eisenhardt 1989). Since only a limited number of interviews could be done, choosing subjects who represent different roles and belong to different organisational units was considered to provide the most extensive and comprehensive material for the analysis.

The data collection instruments, i.e. the interview protocol and the interview framework constructed on the basis of the conceptual schema, were not altered during individual case studies. Some adjustments were done between the case studies in order to include learnings from previous case studies and to adapt the data collection instrument to the specific features of the case.

However, the experiences from the first two case studies showed that the interviewing technique had some weaknesses in studying the work flows and processes related to change management. The interviewees found it difficult to describe their actual work flows in words. They very easily drifted into describing the official processes (see Figure 6) stated in the process documentation, or if they were involved in process improvement tasks, it was natural for them to describe the processes as they thought the processes should be performed. Therefore, there was a danger of getting a description of the official process or

the prescriptive process, not a descriptive process model of the current situation (see Figure 6). For this reason, triangulation was used in the third case study. The interviews were complemented with other methods, which gave additional information and indications of the actual processes and work-flows performed by the software designers and other software personnel. The complementary methods were process measurement, analysis of tools related to change management, and document analysis.

Appendices A, B and C give explanatory narratives of the case studies, and only a brief summary of the cases is presented in the body of the thesis to give the reader a quick overview of the empirical data used in the construction phase of the thesis. As Barton Cunningham (Barton Cunningham 1997) states, the problem with narratives is that since they are used to summarise an assortment of evidence, they may not seem to be presented in a standardised structure, and they may appear unfocused. This can also be seen from the narratives presented here. However, the descriptive information is assumed to provide evidence of the practices in the field in an orderly manner.

The interviews were documented as follows:

- Interview notes from individual interview sessions were written by the interviewer.

- An interview narrative was composed from the individual interview reports.

- A change process analysis document was derived by adding to the interview narratives the results of the brainstorming sessions and other discussions which were arranged to discuss the findings of the interviews.

The notes from the individual interview sessions are confidential documents, which were used only by the interviewers. The interview narratives and process analysis documents were distributed to the organisation under study. These are project documentation and cannot be publicly reviewed. Shortened versions of the change process analysis documents of the case studies presented in the appendices A and B have been published (Mäkäräinen 1996). The third case study (presented in appendix C) has not been published before.

2.4 Research methods used in the construction stage

The constructs proposed by this thesis are derived from both the case studies and the literature study. The presented change management models are based on the change management models presented in the literature, which were adapted to the needs and requirements identified in the case studies.

The case data was analysed using within-case analysis. Within-case analyses typically involve detailed case study write-ups for each case study (Eisenhardt 1989). The write-ups were compiled from notes and observations made during the data collection, and took a the form of narrative descriptions of the cases. Brief summaries of the narratives are presented in the appendices of this thesis.

The process models are described primarily in text form, but some graphical process models are provided to illustrate the processes. The graphical process models are presented as data flow diagrams (DFD). The syntax of DFD models is the following (Pressman 1992): (1) the squares represents the external entities, which act as sources of system inputs, or sink of system outputs, (2) the arrows represent the data flows between the activities, and the arrow head indicates direction of data transfer, (3) the bubbles represent the actions which transform the input data into output data, and (4) the double lines with arrows illustrate a repository of data. The DFD notation is illustrated in Figure 3.

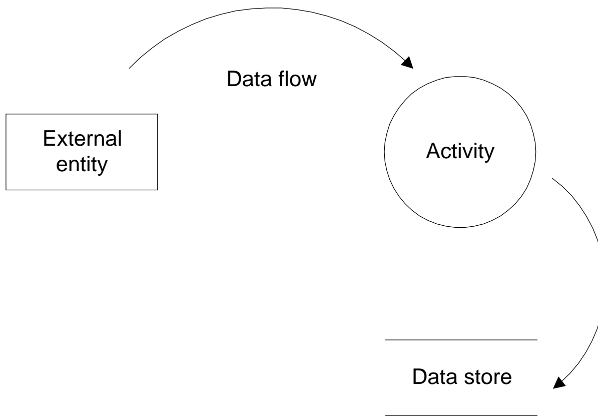


Figure 3. Symbols of the data flow diagram.

2.5 Research methods used in the demonstration stage

The constructed change management model is demonstrated in one additional case study. The model is used in defining the change management processes in the organisation studied and in implementing a new change management solution in the organisation.

The starting point for the case study was a need for implementing new solutions for change management through better understanding of the initial status of the change management processes. The trigger for the project came from the deployment of a new SCM tool, which has features for supporting change management. The aim was to examine the possibilities of using these features for better change management support in the organisation.

The case study follows the Pr^{2}imer cycle. The Pr^{2}imer service package has been developed by VTT Electronics (Technical Research Centre of Finland) for supporting actions that aim at improving software processes (Karjalainen et al. 1996). The Pr^{2}imer method divides a process improvement project into the four phases illustrated in Figure 4.

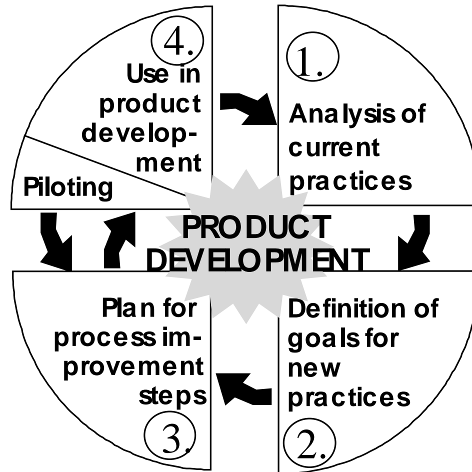


Figure 4. Pr^2imer process improvement cycle.

The goal is to first analyse the current practices in order to achieve a better understanding of the current processes, problems and strengths in change management in the case organisation. Based on the results of the first phase, a decision whether the organisation wishes to change its current way of managing changes is made. The second phase defines the goals for the new practices to be adopted to change the current situation. The third phase aims at planning the transition from the current situation to the target state. The final phase includes the piloting and deployment of the planned improvement actions. Chapter 7 presents:

- a description of the initial change management status analysed in phase one,
- a description of the target state of change processes defined in phase two,
- a description of the implementation solution defined in phase three, and
- a summary of experiences from the piloting and deployment of the new change management solution in the organisation which was performed in phase four.

2.5.1 First phase: Analysis of current practices

The goal of the first phase was to describe the initial status and problems in the change management practices in the case organisation. The goal was to gain familiarity with the change management environment in the case organisation, apply the problem classification framework presented by this thesis, and define what kinds of improvement needs can be identified in the change management processes. The methods chosen were:

- semi-structured focused interviews,
- analysis of tools used in change management or in interfacing change management tools, and
- document analysis.

Semi-structured focused interviews were selected as the main research method for defining the initial status of change management in the organisation. The focus area was software change management, and the goal of the interviews was to identify and characterise the change management processes and find the problems related to change management. The conceptual schema used in deriving the interviewing framework is the same as in the background case studies (see Figure 2). The schema defines the rough focus areas of the interview sessions, and the framework adapts the common schema to the special features of the organisation in question. The interview framework was used as an interviewing instrument in formulating specific questions and guiding the discussion. The interviews were conducted by interviewing each interviewee in a separate session, i.e. only one interviewee was present at each interview session.

Since the purpose of the study is to learn about change management processes, not to test hypotheses, no coding of the interview material was done (Glaser & Straus 1967). The interview results were analysed throughout the process. The understanding of the processes and problems related to them started to take form immediately after the first interview sessions. Triangulation (Barton Cunningham 1997), i.e. combining the perspectives of the interviewees together with the results of the change management tool and document analysis, was used in creating a narrative of the case study.

A similar division for maintenance-related questions has been presented by Layzell and Macaulay (Layzell & Macaulay 1994). They used the following five sections: (1) Background and overall structure of an organisation, (2) Structure and composition of the maintenance function, (3) The maintenance activity, (4) Assessment of the maintenance function, and (5) Future maintenance requirements.

The original purpose of the framework was to support requirements elicitation for the development of an application management environment (Mäkäräinen 1996). The idea of using focused interviews in defining the strengths and bottlenecks in processes was later used in defining the Pr^{2}imer process improvement service package. An expanded interview framework and focused interviews have been used in several successful process improvement programmes in co-operation with VTT Electronics and Finnish industry (Mäkäräinen & Komi-Sirviö 1996, Karjalainen et al. 1996).

The subjects within the case organisation were selected using theoretical sampling from different types of projects (user interface projects, hardware interface projects, etc.) and different software development tasks (project managers, inexperienced designers, senior designers, persons responsible for configuration management, etc.). The subjects were not given any information about the interview focus area in advance. There were two interviewers, the author of this thesis and a member of the software process team of the organisation. The author of this thesis performed a document study to get familiar with the organisational structure and organisation-specific practices and guidelines. This ensured that the interviewer spoke the same language as the interviewees, and also saved time in the interviewing sessions, as the interviewees did not have to elaborate on basic issues concerning the tools used organisation-wide, responsibilities within the organisation structure, etc. The objective of the interviews was clearly stated to the interviewees before the interviewing sessions; i.e. they were told that the goal is to define the current status of the change management practices in the organisation in order to identify their strengths and weaknesses, and to examine if the new software configuration management system could be used more efficiently in supporting the change management processes.

The individual interview sessions lasted from one hour to three hours. As interview sessions usually took place in a meeting room, the interviewees had no

possibility for checking actual software documents during the interview sessions. In between the interview sessions, some of the interviewees showed examples of relevant change documentation to the interviewers.

Since the first two organisations studied were rather small, practically all software designers were interviewed during the studies. The third company was considerably larger, and only a part of the software personnel could be interviewed (six persons).

The interpretation of the interview results was done by the author of this thesis. The interpretation was done in three phases. First, the results of individual interviewing sessions were written down as individual memos. The memos were sent to the interviewees, who could read and check them for any misunderstandings or add comments or information they had forgotten to mention in the interviewing sessions. The individual interviewing memos are confidential, and were not submitted to anyone else but the interviewees. As the interviews continued, a summary of all the interviews was constructed. The summary report was sent to the interviewees, the case company representative in the interviews, and the manager of the SCM tool deployment project for comments. A review was arranged, where the author of the thesis presented a summary document and the reviewers could give their comments. The comments were discussed in a meeting, and the changes agreed to be relevant were made.

Goguen and Linde (1997) give a guideline for researchers performing interview studies:

Do not ask people to describe activities that they do not normally describe, or if you do, then do not believe the answers.

The focused interviews on change management could not avoid asking people to describe activities which they normally do not have to express in words in normal conversation. This problem was tackled by examining the recorded data related to change activities, and comparing them with the interview results. For example, the error recordings were browsed to study the error handling processes, and software item version histories were examined to find information about traceability issues.

The interviews are complemented with tool and document analysis. Both were done prior to the interviews to familiarise the interviewers with the change management environment used by the interviewees. Also, some additional documentation could be identified during the interview sessions, so the document and tool analysis were extended to continue throughout the whole first phase of the Pr²imer cycle. The purpose of the tool analysis is to identify the tools currently used in change management, find their strengths and weaknesses and study the change histories recorded using them. The purpose of the document analysis is to study the documentation related to change management, which mostly consisted of official process descriptions and user manuals.

2.5.2 Second phase: Definition of goals for new practices

The second phase of the Pr²imer improvement cycle generates the definition of new change management practices based on the problems and improvement ideas identified during the first phase of the cycle.

The new processes are defined using the generic change management processes described in chapter 6. The process descriptions are defined by a small group of software personnel from the organisation and the author of this thesis. The process for managing requirement-level modifications is further studied in a separate project, since it requires a more thorough analysis across organisational entities outside the software department. The group involved in the process description includes members from the software method and process improvement group. The resulting processes are reviewed by software engineers from product development projects.

The change processes are described using the Information Mapping method (Horn 1992). The method provides guidelines for structuring and presenting infromation. The Information Mapping method has been in use since 1972 mostly in documenting training and procedural and reference manuals. It supports both paper and online modes. The process descriptions are aimed to be included in the company's online process documentation, so support for the online mode is required. The method supports separating information into small units based on its purpose or function for the reader. The method aims at supporting the readers in quickly finding the relevant information in the document.

2.6 Evaluation of the results

The main part of the result evaluation is concerned with the evaluation of the success of the demonstration phase, i.e. how the change management model is instantiated in the organisation used in the case study. The following methods are used:

- Discussions with the development team.
- Observations in the case organisation.
- Interviews of users who had been involved both with the old change management system and the new one.

The discussions with the team that is responsible for the instantiation of the new change management solutions in the organisation provide an evaluation of how the development team considered the new solution to address the change management problems and needs stated. The team has a deep understanding of the initial goals and problems, since it is involved in defining them together with the software practitioners. On the other hand, since the development team defines the new solutions by themselves and they are not dealing with the change management problems in their everyday work, their evaluation may be biased and not provide actual improvements in software development. Therefore, the qualitative evaluation by the development team is complemented with observations and interviews of actual end users.

The observations were made by studying the actual change management system and databases, and how they were used in the projects. Since the observations were made by the author of this thesis, who was a member of the development team, the observations are subject to the same bias as the discussions with the development team.

Some end users, who have been involved with software development tasks long enough to use both the old change management system and the new solution, were interviewed to get a more objective evaluation. The interviews are structured into two parts, evaluation of the performance of the new system with re-

spect to the problem areas, and evaluation of how the implementation solved the specific problems stated at the beginning of the case project.

The main success criterion for evaluating the results is to prove that the constructs built in this thesis were usable and helpful in defining the new change management solution in a company developing embedded software, and to evaluate if the new solution was successful in solving the change management problems found in the organisation studied and if it fulfilled its function in the everyday software development work in the company.

The constructed change management model is assessed through studying its instantiation in one case study. The construction is assessed against the criteria of value. March and Smith (March & Smith 1995) advise the assessment of the product of design science to answer the questions "Does it work?" and "Is it an improvement?". The work presented here is a representative of design science, where the attempt is to create things that serve human purposes, as opposed to natural sciences, where the focus is on explaining how and why things are. (March & Smith 1995)

The first question, "Does it work?", is examined by observing the instantiation of the model in the case study, and the evolution and usage of the instantiated model in the organisation. The observation methods include discussions with and interviews of people who participated in the instantiation, use the instantiated system and have been responsible for the support of the system when it has been used, and studies of the maintenance histories, as well as development and maintenance documentation. Also, the actual change data collected in the company was observed to evaluate the system.

2.7 Research process

The author started exploring the subject of software maintenance in the AMES project in 1993. The initial goal was to focus on the software maintenance phase, i.e. the activities taking place after the delivery of the complete software system. The author was responsible for analysing the maintenance activities in the two companies in order to derive the requirements for the maintenance support environment, including process, method and tool support. The analysis

work was done in co-operation with the software designers and process improvement personnel in the organisations. In the first case organisation, the analysis team consisted of the author, one senior researcher from VTT (the employer of the author at that time), who had an extensive background in software maintenance and configuration management issues, and a representative from the case organisation, who had process improvement responsibilities in addition to working as a software designer in the project analysed in the case study. The second case study was done by a team of two persons, the author of the thesis and a representative of the company studied, who was responsible for software process improvement and worked as a software designer in the company. The third case study was done later outside the scope of the AMES project. The study was performed by the author of this thesis with the aid of two members of the software process team at the company, both responsible for software configuration management processes and tools.

As the work progressed, the focus of the study shifted from the software maintenance phase, i.e. the post delivery activities, to software change management, i.e. change activities at any point of the software life cycle. These three case studies together with a literature study formed our understanding of software change management as it is defined in this thesis. After the first two case studies performed in the context of the AMES project, the author reported the results at that point in a licentiate thesis (Mäkäräinen 1996).

These three case studies (described in appendices A, B and C) are presented in this thesis as background information for the change management models presented. This means that these case studies are part of the problem analysis phase of the research. The narratives of the case studies are presented in the appendices, and a brief explanatory summary of the cases is given in chapter 4. The literature study constitutes another part of the problem analysis phase. The literature study was primarily done simultaneously with the two first case studies, but has been extended to continue throughout the whole research period.

The fourth case study presented in this thesis demonstrates the constructed change management model in one organisation. This organisation is not one of the three used for constructing the change management model. The fourth case study was started in the same manner as the first three case studies were performed; i.e. by analysing the change management practices in the case organisa-

tion. The analysis was done by a group of two persons, the author of the thesis and a representative of the company, who was a member of the software process team. The results of the analysis phase were used as requirements for instantiating the model. The new change management processes were instantiated from the generic model by the author of this thesis, with the exception of one process instance, which was instantiated in a separate project by the process improvement team of the company. The supporting environment, piloting, training and deployment of the new system were done by the process improvement team of the company with the help of subcontractors in implementation matters. However, the author of this thesis followed the work closely, and acted as a consultant throughout the implementation and piloting periods.

2.8 Summary

The research approach adopted in this thesis is constructive. The constructs proposed are derived using a literature study and three case studies, and evaluated in one case study. The main instrument used in the case studies is focused interviews of the software personnel in the organisations studied.

3 Analysis of related work

3.1 Overview

This chapter presents an overview of the research related to software change management. It discusses the relationship of software change management to related concepts. The chapter continues with a discussion of the software change management problems presented in the literature. The findings from related work are later used in chapters 5 and 6 for defining how software change management processes and related problems are defined in this thesis.

3.2 Scope of software change management

The processes and problems related to changing software items have traditionally been related to the maintenance of the software (e.g. IEEE 1993, Schneidewind 1987, Bjerknes et. al. 1991). Controlling changes during the software development time has been defined as a task for software configuration management (SCM) (Pressman 1992). This thesis combines these two viewpoints to software change management: it examines the processes for managing changes to existing software items and configurations in any phase of the software life cycle .

Taramaa (1998) has discussed the relationship of software configuration management and change management. His software configuration management framework examines software configuration management from the viewpoint of version control, release-oriented and change-oriented SCM. The framework presents a 12-level improvement framework for software configuration management. The first levels include version control oriented activities. At these levels change management is limited mainly to the creation and storage of change documents. The levels from four to five have characteristics from software manufacturing and also include some aspects of change control, such as a link between the change documentation and the changed software items. The sixth level is called the "Change-oriented level", and it emphasises request-driven change control, including support for software evolution and mainte-

nance. The last six levels include product management components, highlighting efficient usage of reusable components and advanced assembly systems.

Weiderman et al. (1997) distinguish software maintenance from system evolution using the following criteria:

- Software maintenance deals with fine-grained, localised changes, while system evolution deals with coarser-grained, structural changes. System evolution changes the structure and architecture of the system, while maintenance changes leave the structure of the system relatively constant.

- Software maintenance is a short term activity, which produces few economic and strategic benefits. System evolution on the other hand increases the strategic and economic value of the software.

- Software maintenance typically responds to one software requirement at a time, while system evolution allows the system to comply with broad range of new requirements.

In this thesis, software maintenance refers to the software life cycle phase beginning when the first delivery of the software is made, and ending when the software is taken out of use. Evolution, on the other hand, refers to the stepwise, incremental development of the software during its lifetime. Evolution of the software system takes place both in the development and maintenance phases through successive and concurrent changes. The activity of managing these changes is called change management. Evolutionary software development is a process in which the software is delivered incrementally. The feedback and analysis of the latest incremental delivery generates the requirements for subsequent deliveries. (Lam et al. 1999)

The definition of change management is derived from the discussion of software maintenance, evolution and iteration presented by Schneidewind (Schneidewind 1987). Schneidewind asks the following questions:

- Do we have the wrong model for maintenance? He states that change activity and change management should be an integral part of development and

all other phases of the software life cycle. The change should be associated equally with post-delivery activities and with development.

- Do requirements end in the requirements phase? Schneidewind concludes that requirements change continually, and it is not possible to develop a complete, consistent and unambiguous specification prior to software design. The major problem in requirement management is the evolution of the requirements in response to changes in the environment.

- Is the life cycle model appropriate for maintenance? Schneideman criticises the association of software maintenance only to the post-delivery phase of the software life cycle.

As Lehman suggests, change is intrinsic in software and must be accepted as a fact of life (Lehman 1980). The relationships between the concepts of software development, maintenance and software change management, as they are defined in this thesis, are illustrated in Figure 5. Software change management refers to the management of the evolution of the software. Evolutionary software development consist of stages which incrementally expand the software under development (Boehm 1988). The software evolution starts right from the beginning of the creation of a software item. There are two main reasons for changes during the development time: the requirements of the system change during the development time, and the development of a complicated system is very error-prone (Ince 1994). The more complete the software becomes, the more time and effort is required for change management. In the testing and maintenance phases, practically all the development work is change management. The development of new software mainly takes place at the beginning of the software life cycle and decreases towards the implementation, while evolutionary changes increase. The situation will be even more dramatic when the development project bases the development work on reuse of software components or adaptation of an old system to new requirements. In these cases, the development project may focus entirely on managing the changes to reused items instead of creating new software from scratch. The evolutionary development style also shifts the nature of the development work from waterfall type of development to managing the evolution of the software. For example, the development model defined for developing fuzzy logic control systems (Isomursu 1995) proposes controlled evolution as the main vehicle for deriving fuzzy sys-

tems. The vagueness or uncertainty of the processes typically controlled by fuzzy control systems makes exact specification of the fuzzy design parameters difficult, if not impossible, without exploration. Therefore, iterative prototype-based development has proved to be successful in deriving fuzzy design parameters.

The target of this thesis is to achieve a better understanding of change management problems and processes, during both the software development and maintenance phases.

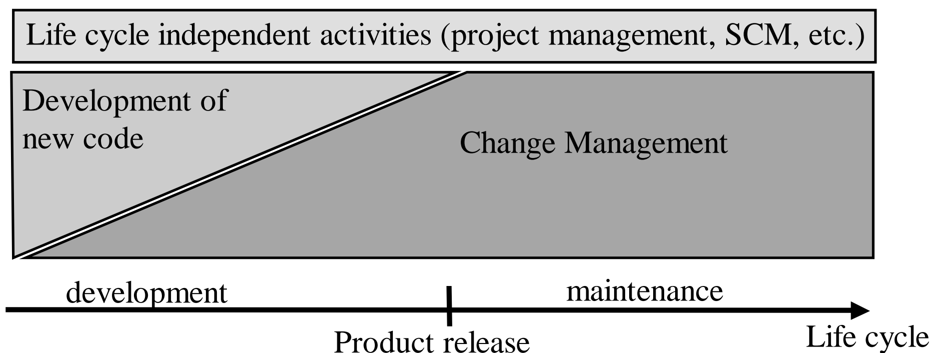


Figure 5. Software change management in software development and maintenance phases.

Figure 5 illustrates the relationships between some key concepts of this thesis. The software life cycle is divided into two main phases: development and maintenance. The maintenance phase starts when the product is first released, and the development phase includes all activities before that. In some cases, the maintenance phase may not exist at all. This is the case described in the first case study presented in this thesis (Appendix A). The actual software work is characterised as development of new code and managing changes to existing code. The amount of new development usually decreases towards the end of the development phase, and in the maintenance phase all work consists exclusively of managing changes to the existing system. The proportional share of new development and change management vary between cases. Sometimes the life cycle starts from a situation where everything has to be built from scratch. In these cases, the proportion of new development is 100% at the beginning and decreases towards the end of the development phase. In some cases, the devel-

opment phase starts with reusable items from previous development tasks, and the proportion of change management is high right from the beginning of the development phase.

In this thesis, the term "software maintenance phase" is used to describe the post-delivery phase of the software life cycle, and the term "software change management" to describe the management of modifications to existing software items, regardless of in what phase of the software life cycle the modifications are made. Several definitions for software maintenance have been presented in the literature. The definitions can be categorised as follows:

- *Life cycle approach.* Software maintenance is the phase beginning after the delivery of the software and ending at the retirement of the application.

- *Life cycle approach combined with functional description.* Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment. (IEEE 1993)

- *Plain functional approach without restrictions concerning the life cycle.* The job of software maintenance is to correct errors and to change program operation as requirements change. (Berns 1984)

The major difference between software and hardware maintenance is that software, unlike hardware, does not wear out in use. The maintenance of the hardware parts of a system includes replacement or repairing of worn or broken parts, while software does not need this kind of sustained repair.

In addition, the word ‘maintenance’ is generally used by the software engineering community to point to all activities taking place after the first delivery of the system (Turski 1981). As Sommerville stated, the addition of a new wing to a building would never be called ‘maintenance’ by construction engineers, yet adding a totally new feature to a software system is considered a maintenance activity (Sommerville 1982).

However, software starts to evolve and require changes even before its maintenance. Software change management is an integral part of the software devel-

opment process from the very beginning. The environment where the software is supposed to operate evolves thoroughout the system development time, creating change pressure on the original requirements (Lienz & Swanson 1980). Furthermore, the software items in different abstraction levels require modifications because of the defects introduced by the software engineers and the increasing knowledge about the problem domain. This thesis defines 'software change management' as a discipline for managing the changes to the software system thoroughout its lifetime, in both the development and maintenance phases (Schneidewind 1987) (Lehman 1980). It claims that there is no fundamental difference in, for example, correcting an error in the software a week before the release date during the development phase, or a day after the release during the maintenance phase. Differences may be found from contractual and organisational viewpoints, but not in the procedures carried out to change the software.

Swanson also defined the most commonly referred to classification of software maintenance tasks (Swanson 1976). The classification covers four dimensions of software maintenance tasks: corrective, adaptive, preventive and perfective changes. The diagnosis and correction of errors found in operational software are characterised as corrective changes. Adaptive changes aim at adapting the software to environmental changes, such as hardware changes or a new operating system. Preventive modifications are made to anticipate and prevent future problems, e.g. through re-structuring existing systems. Perfective modifications alter the functionality of the software, usually by adding new features. Although Swanson discusses software maintenance tasks, the experiences from the case studies presented in this thesis show that the division is applicable for changes taking place both before and after software delivery.

In addition to the close connection to software maintenance research, software change management is very tightly related to software configuration management. Software configuration management is defined by the IEEE 828 standard (IEEE 828 1990) to include the following basic elements:

- Configuration identification. Identifying and defining the configuration items for the product.

- Change control, including configuration control. Controlling all changes to these items throughout the life cycle of the product.

- Configuration status accounting. Recording and reporting the status of configuration items and change requests.

- Configuration audit. Verifying and auditing the completeness and correctness of these items.

As Taramaa (Taramaa 1998) summarises, the traditional definition presented above has later been extended to cover issues related to manufacturing, process management and teamwork (Tichy 1988, Dart 1991). Software change management, as defined in this thesis, covers the issues related to managing changes and change requests, and does not examine the problems of version control, building baselines, manufacturing products, etc.

Software changes can be divided into active change management, reactive change management, and pro-active change management (Ackoff 1967). Active change management deals with planned changes. The desired change in the environment is planned, and the software modification is done to support the anticipated change. Reactive changes are initiated by the changes in the environment the system operates in. The environment has changed somehow, and the system has to be modified to react to external changes. Pro-active changes anticipate future changes, and aim at making the system fit for easy reactive changes (Michelis et al. 1997).

Change is a natural aspect thoroughout the software life cycle and it has to be accepted as a fact of life. (Lehman 1980, Bersoff & Davis 1991). Software requirements change and evolve (Lam et al. 1999), and software developers introduce defects. Producing a new design is always open-ended, changes to initial designs occur inevitably because of changes in the environment (Brown 1993) and as more is learned about the problem domain. This is not a special feature of the software engineering discipline, it also happens in other engineering disciplines. Examples of the unpredictability of the design work in other engineering disciplines are for example the design of the Concorde or the Channel Tunnel (Bennet 1996). Hence, software undergoes changes throughout its lifetime (Lehman 1998).

The majority of software personnel today work in maintenance and evolution of old software as opposed to the development of new software. (Lientz & Swan-

son 1980), (Nosek & Palvia 1990) This stems in part from the following considerations:

- The lifetimes of the applications are long. The maintenance phase is often much longer than the development phase. The systems require constant support and expert knowledge to keep them running. When the application lifetime is long, the original requirements are prone to change over time, creating pressures for modifications to keep the system valid. The life expectancy of software has been proposed to be proportional to size, i.e. large applications tend to be in use for relatively long periods of time, while small software systems often disappear from use in a shorter time. (Jones 1989)

- New systems are often derived from existing ones. The borderline between software development and software maintenance has become blurred, since the development projects may be based on an old software baseline, which is adapted to meet the new requirements. Therefore, software developers actually implement changes instead of designing new software.

3.3 Why is software change management difficult?

A number of reasons for the difficulty of software change management can be found in the literature. These include: age of software, loss of design knowledge, loss of original requirements, accumulation of problems and change needs, lack of design for change, time pressure, diversity of tools and information, poor image of change function, poor maintainability of just released software, code decay, few tools and methods, verification across several product versions, and focusing on developing new software instead of managing existing systems. The problems are examined in detail in the following subsections. The problems found during the literature study will be complemented in chapter 4 by an analysis of the problems identified by studying the change processes in companies developing embedded software.

Many software systems are old (Osborne & Chikofsky 1990). They have been designed and implemented using outdated methods and tools (Schneidewind 1987). The original developers of the system have seldom even known how long the system will be used. Examples can be found in systems which were not de-

signed to be able to operate beyond the year 2000 because the original developers could not anticipate such a long lifetime for the system.

The design knowledge and the rationale behind the design decisions have disappeared (Rugaber et al. 1990). No-one knows precisely what the software actually does and why. The person changing the software often has to deal with incomplete or outdated documentation. In the best case the design decisions used are documented, but the design rationale behind the design decisions is very seldom recorded (Abbattista et al. 1994).

The original requirements have been lost. Software requirements are forgotten, and therefore often violated when the changes are implemented (Parnas 1996). The original requirement specification documents may have been incomplete to start with, often leaving out requirements which seem trivial at the time of defining the software requirements. Moreover, the requirement specification document may have become inconsistent with the actual system implementation during the evolution of the system, when new requirements have arisen or requirement changes have been made, and the related requirement specifications have not been updated.

The problems and change needs tend to accumulate. An implemented change prepares the way for the introduction of another (Pressman 1995). The error fixes create new errors. The error fix may be done in a hurry and with inadequate resources, causing deficiencies in impact analysis and regression testing. Also, the quality of the system may deteriorate because of changes over time, resulting in more problems in understanding and modifying the system in the future (Brown 1993).

The software is not designed to be easily maintained or modified (Capretz & Munro 1994) (Brown et al. 1995). The quality requirements set for a system rarely have specific requirements concerning maintainability issues. As the primary goal of the development phase is to release the system fulfilling the user requirements, the maintainability requirements are often considered to be of secondary importance.

Often the modification has to be done as quickly as possible. There is no time to think about the quality or the impacts of the change (Brown et al. 1995). The

modification activities interrupt the development work, and if they are not planned, they also cause delays in project schedules (Genuchten et al. 1992).

During the software life cycle many different tools are used, and usually each tool manages only a subset of software-related information (Cutillo et al. 1996). This results in problems for software engineers in finding relevant information when managing changes to the system, and keeping the system consistent when developing and changing the system (Ketabchi & Sadeghi 1996).

The image of software change activity is not highly valued (Lano & Haughton 1993) (Kellner 1993). Often new programmers are assigned to maintenance tasks to 'learn' about the application domain and how to program. The changes are regarded as unplanned and unwanted tasks.

The maintainability of just released software is poor (Brown et al. 1995), and it gets worse during the maintenance history of the system (Jones 1989). It is too late to think about maintainability only after the software has been designed and delivered (Schneidewind 1987, Capretz & Munro 1994). Maintainability must be built into the system when the initial design and implementation decisions are made. Maintainability and modifiability should be planned and designed right from the beginning of the development work. The early design decisions are found to have more impact on software maintainability than the implementation algorithms (Rombach 1987). Also, often the changes made to the system over time gradually degrade the system, decreasing its maintainability (Jones 1989) (Schneidewind 1987). Without proper management and attention to the quality of the software modification activities, the quality of the software deteriorates over time. For example, the complexity of the source code may increase (Bennet 1993) and the documentation may become inconsistent with the source code (Briand et al. 1994, Brown et al.1995).

Few methods or tools for supporting software modification and change management are in active use in companies (Layzell & Macaulay 1994). Several critical and laborious tasks, such as impact analysis and consistency checking, lack advanced tool support and usually have to be performed manually. During the last decade a lot of tool support from the research community has emerged (Kellner 1993), but it has not yet been effectively adopted by the industry (Chapin 1993, Brown et al. 1995).

The verification of a change is complicated, when the changed software part is shared by several products or product versions. The change has to work in all products in which the new software will be used. (Bergey et al. 1998) This problem is typical in companies who use the product line approach, where core components are used by several product projects. Propagation of changes made to core parts to multiple deployed products in the product line is challenging. (Bergey et al. 1998)

The main focus of software engineering research has been in software development (Ward 1993). Most of the research done in the field of software engineering deals with methods, processes and tools for the development of new software, and largely ignores the software evolution and maintenance viewpoint (Schneidewind 1987). Advanced techniques exist for designing new software systems and forward engineering activities, but these techniques often neglect important aspects affecting the maintainability of the system, for example maintaining consistency between work products, defining and updating links between semantically related items, etc.

3.4 Software change management – Product dimension

This thesis studies change management in the context of embedded software. This subchapter discusses the special features of embedded software which create special needs and requirements for software change management. The following special features are examined:

- Concurrent system engineering
- Sharing of software parts in several products or product families
- Primitive software engineering environments
- High reliability demands

These features will be further discussed in the following sections.

During the development of the hardware and software components, the requirements change and evolve. The changes relate to all parts of the product, creating problems and special requirements for the management of change. Concurrent hardware and software development adds one more variable from the software viewpoint. Not only does the environment where the product will operate evolve during the software development time but the hardware environment it is going to operate evolves, as well. In addition, the complete hardware environment is not available until the late integration testing phases (Mittag 1996, Taramaa 1998) creating problems in testing, location of problems and verifying interface requirements.

Customer-specific product features are often implemented by the means of software rather than hardware. Using a common hardware environment and specialising it with software is usually more cost-effective than designing and producing several hardware versions. Managing several customised product versions and propagation of modifications among the versions results in change management problems (Bersoff & Davis 1991). In addition, the new products are often based on the software components or product baselines developed in the previous projects, as for example in the case presented in appendix C. The nature of software development is to change the existing software to meet new requirements, adapt it to a new hardware environment and add new features by making changes to old system baselines instead of writing software from scratch. (Vierimaa et al. 1998)

The development environments and methods available for designing software systems embedded in electronic products are often rather primitive (Vierimaa et al. 1998). The developers have to design time, memory or power consumption critical parts using assembly language, because the processor vendor does not provide compilers for the high level languages, or the code generated by the compilers provided is not efficient enough to meet the time, memory or power consumption requirements set for the software. Even if the hardware vendors do provide compilers for higher level languages, support for other software development activities, such as testing, code measurement, etc. is seldom available. A recent survey (Seppänen et al. 1997) states that as many as 45 % of companies developing embedded software use a combination of the C language and processor-specific assembly languages. This has two major effects on change management; the source code is hard to understand, and automated support for

quality monitoring, consistency checking etc. is difficult to provide (Vierimaa et al. 1998). The tight software–hardware relationship also results in a difficulty of analysing the software as a separate entity. For example, there may be problems in the system which are not solely hardware problems or software problems, but are a result of the hardware and software not working together (Ojennes 1998). The analysis of the change requests in one case project in the domain of embedded software made by Stark et al. (Stark et al. 1994) revealed that only 280 problem reports out of 982 resulted in a software or hardware modification. The rest of the problems could not be repeated (317 problem reports), were duplicates (185 problem reports), or were caused by a human error, configuration limitations or other, misclassified reasons.

The reliability demands for embedded software are often high (Seppänen et al. 1997). Changing software after its delivery is generally very difficult, and sometimes even impossible, for embedded software (Taramaa 1998). The product in which the software is embedded may operate in an inaccessible environment. The volume of products released may be huge, making after-release corrections to products that have already been delivered very difficult and costly. Therefore, it is important to assure that the software does not need to be modified after the delivery, and if such a need arises, the modification does not cause new problems.

3.5 Software change management – Process dimension

One solution for achieving better control over software processes is defining processes for supporting the actual work-flows and activities taking place when people work (Zahran 1997). Two main factors of successful change management are communication and control (Bersoff & Davis 1991). Formal and supported change management processes can be used as a tool to achieve better control over the process (Pressman 1995) and to support communication.

The types of processes and process models are illustrated in Figure 6 taken from Bandinelli (Bandinelli et al. 1995). The figure also illustrates the problems faced in understanding software processes and creating process descriptions. The two sources for examining software processes are observation of actual processes and analysis of official process descriptions. The official process de-

scriptions are *prescriptive* process models, i.e. they describe how the processes should be executed. The deficiencies of the process description methods and languages and trade-offs caused by practical limitations, such as lack of resources, change resistance, etc. result in variations between the desired process and the official process. In order to understand the actual processes in an organisation the process agent aims at composing a *descriptive* process model, i.e. a model which describes the software processes as they are executed. The process agent faces the problems of poor visibility and traceability of the software processes. Software processes are very abstract by nature, and it is difficult to study them without introducing bias.

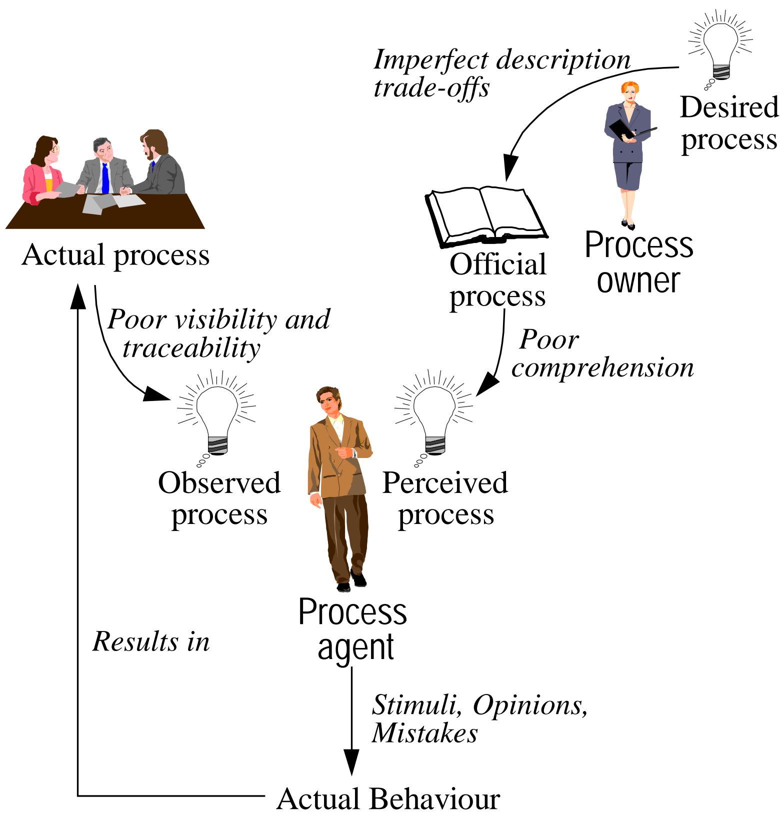


Figure 6. Types of processes (Bandinelli et al. 1995).

As Boehm (1988) points out, the very first software process models (Benington 1956) did not address change during the development time. The models stipulated that software can be developed in successive stages, and each stage could be completed before moving to the next stage. The waterfall process model (Royce 1970) introduced feedback loops between states, and recognised a maintenance phase after the delivery of the software (Taramaa 1998, Edelstein & Mamone 1992). Still, iteration between successive development phases was undesirable, and was considered to be caused by design errors or incomplete work made in the earlier phases. Later, when development models such as the spiral type (Boehm 1988) and the prototyping process model (Pressman 1992) emerged, the changes during the development time were considered as positive, natural phenomena by the software process models. It was recognised that effort should be directed not only to activities aiming at reducing or preventing changes from happening, but more importantly to activities supporting change management and implementation.

3.5.1 Olsen's change management model

Olsen (1993) presented a change management model which views the whole software development process, including both development and maintenance phases, as a dynamically overloaded queue of changes. His model views all work done by software designers as changes. A change is defined as anything that might require work to be done: development of new features, filling out project management related forms, correcting errors in software, etc. Olsen's change management model is illustrated in Figure 7.

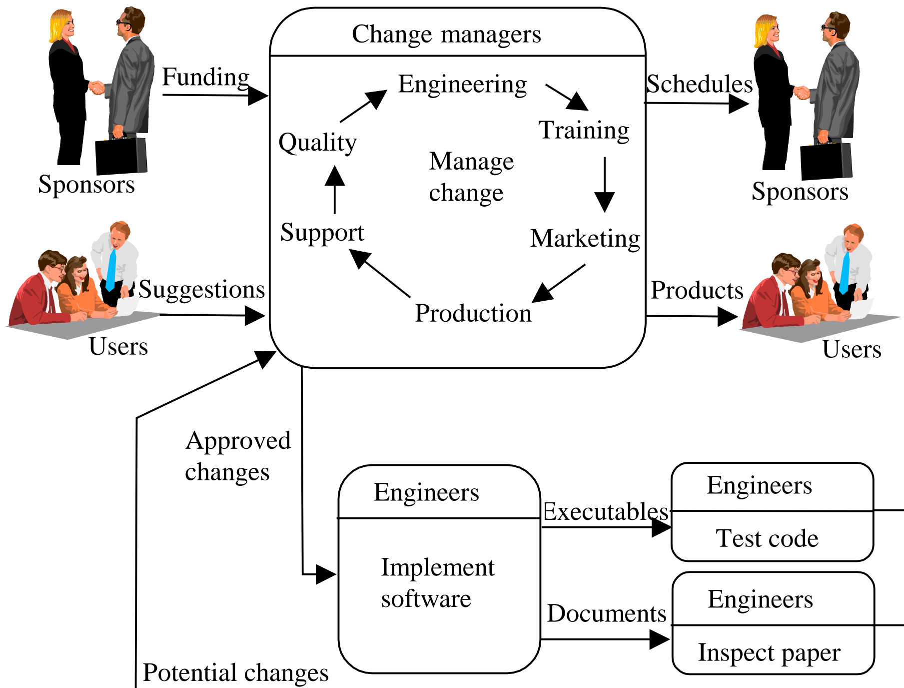


Figure 7. Olsen's change management model.

The model is an abstraction of the software process, where all activities are treated as changes. Therefore, it is not life cycle dependent, and can be applied in both software development and maintenance. The model does not make a distinction between different types of changes, or how different types of changes are managed. It defines the change activities on a very high level; including change creation ("Manage change" in Figure 7), change implementation ("Implement software" in Figure 7) and verification activities ("Test code" and "Inspect paper" in Figure 7). The change sources defined by the model are suggestions from the users, change proposals from verification activities, and change proposals generated by the change managers. The sponsors are treated as a source of funding and as the body monitoring the schedules.

Olsen's model points out the fact that change is a key element in software development. However, by treating all activities as changes, it loses the ability to examine and describe the specific problems and features of changing software items instead of developing new ones.

3.5.2 V-like change management model

The V-like change model describes the technical activities for implementing a change. The process (see Figure 8) is the same for all change types. The following types of change activities are considered (Harjani & Queille 1992):

- *User support*, which includes activities for providing answers to users' information requests and correcting misunderstandings.

- *Corrective changes*, which aim at fixing an error in software without making any changes to the requirements.

- *Evolutive changes*, which include activities for adding new functionalities in response to new or changed functional requirements.

- *Adaptive changes*, which adapt the software to changes in the operational environment.

- *Perfective changes*, which aim at improving non-functional requirements, such as execution time.

- *Preventive changes*, which improve the maintainability of the software and prevent problems in future change activities.

- *Anticipative changes*, which anticipate future problems and aim at changing the software to be robust or easy to modify if the changes are realised in the future.

The V-like change model is designed only for the maintenance phase. It assumes that the change process takes place in an environment where the user already uses the software or the product, and the modified software has to be re-inserted to the operational environment after the change has been made. The model is generic in the organisational sense, i.e. it needs to be instantiated to the needs and requirements of the organisation using it. The same model is planned to be used for all change types. However, the example given by Harjani & Queille (1992) defines two variants of the model for different types of changes. The exceptional change types are emergency fixes and routine cases. The process

variant for emergency fixes is followed in urgent situations. The process is optimised so that the change can be done within very tight time constraints. A simpler and lighter version of the process was proposed for routine corrective changes, where the solutions are obvious, low cost and do not have large impacts on other product parts or product operation.

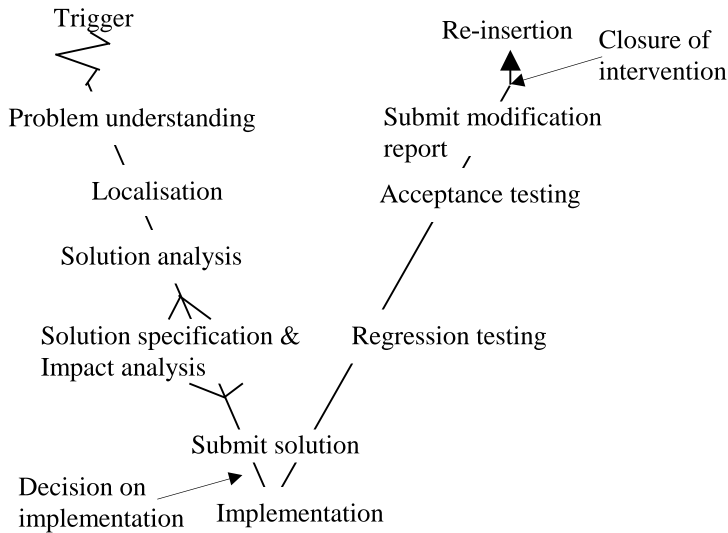


Figure 8. The V-like change process.

The process is started by the receipt of a statement of a problem in the use of the software, or indicating a need for change. The statement can come from external sources or from the maintenance organisation itself. The problem statement or change need is analysed in the problem understanding phase. The purpose is to filter problems and determine the cause of the problem and how it should be processed. The localization activity takes care of determining precisely what is the action requested by the trigger, and which parts of the product will be affected by the change. The solution analysis step generates possible alternatives for solving the problem, and analyses their impacts. After this step, the implementation decision is made. The following alternatives for the implementation decision are given:

- Selection of one implementation alternative.

- Iteration on earlier phases is needed to find more satisfactory solution proposals.

- Abandonment of the request.

If a satisfactory implementation alternative is found, the process continues into the implementation step. The implementation phase is a "mini-development" cycle. The software and associated documents, such as test data and user manuals, are changed using the normal development life cycle used in the company. The modified product is tested using regression testing techniques in order to determine if the new or modified components interact correctly with the unmodified parts, and the behaviour of the product has not changed unintentionally. The acceptance procedures aim at checking that the implemented solution has solved the problem or need stated by the original change trigger. The accepted change is then closed, and the software is re-inserted into its operational environment.

3.5.3 Ince's change process model

Ince (Ince 1994) discusses how software configuration management relates to software change management. According to Ince change management covers change activities during both the software maintenance and development phases. Two main sources for changes are identified: customer requests for new features or error corrections, and the development team for problems identified in the validation phases. Other external sources, such as changes in the hardware environment and the work of product standardization or legislation bodies, or other types of internal sources, such as improvement ideas of software developers, are not discussed by Ince. He proposes that both customer and development team originated changes are managed using the change model presented in Figure 9.

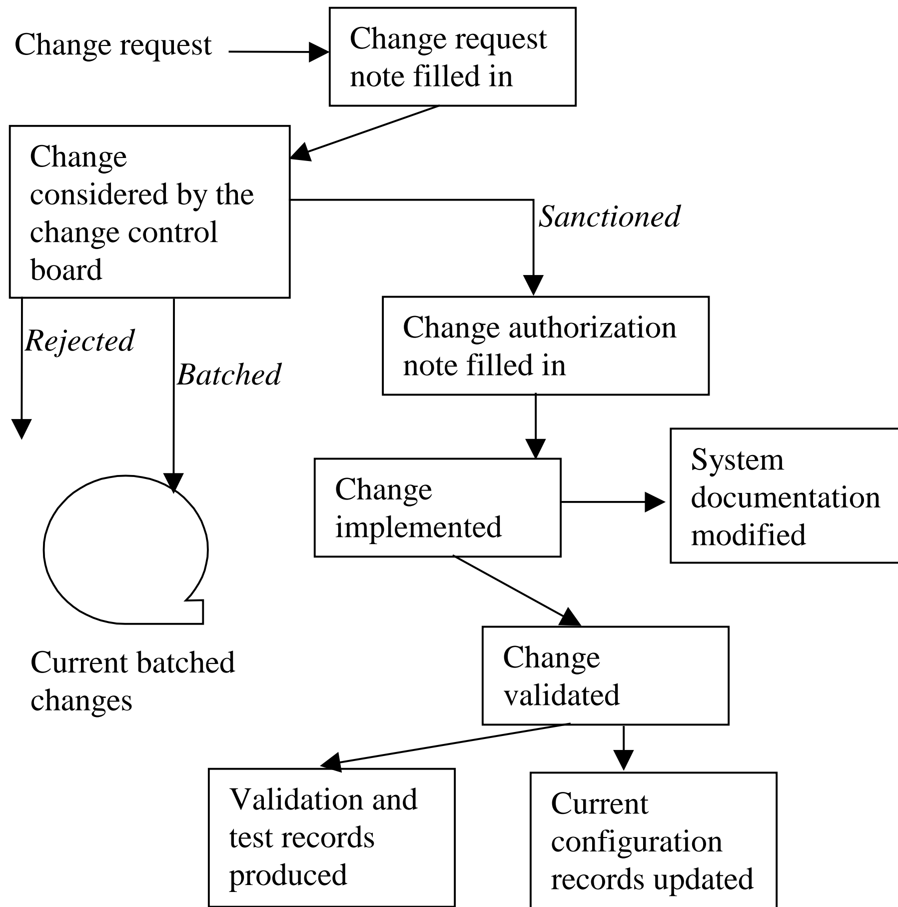


Figure 9. Ince's change process model (Ince 1994).

The model starts with the change request, which is received by the software project. All change requests should be recorded in a change request note. Ince proposes a single change request note template for all change request, even if the two types of changes his model covers (customer-originated new requirements and errors or problems identified either by the customer or the development team) are very different in nature and require different type of data to be recorded and analysed. All recorded changes are then submitted to the change control board. Since the process is the same for all changes, the change control board is also the same for all changes, regardless of whether the change request deals with a request for a new functionality or with a data type error identified

in an inspection of the detailed design document. The change control board may decide to:

- Reject the change, in which case the change will not take place. ("*Rejected*" arrow in the figure)

- Batch the change, in which case the change will take place sometime in the future. ("*Batched*" arrow in the figure)

- Allow the change, in which case the change is implemented as soon as possible. ("*Sanctioned*" arrow in the figure)

When the change is sanctioned, the next step is to document the change in the configuration management documentation, and communicate it to the people responsible for its implementation. The implementation includes changing all documentation associated with the change, and producing new versions of the documents. Once the implementation has been done, the change is validated. Here Ince proposes variation according to the type of change, and suggests the size of the change to be the criterion for defining the validation strategy.

Ince's process relates the configuration management activities, which are configuration identification, configuration control, status accounting and configuration auditing, to the change process presented in Figure 9.

Configuration identification is a process of specification of the components which are placed under configuration control. These components are called configuration items A software change implies changes in component configuration items. Configuration items are identified by a version number. Configuration identification activity takes care of managing the versions and variants of the configuration items.

Configuration control activities take care of communicating the changes to the project, and inform the staff about changes to the configuration items they are dealing with. Status accounting takes care of the recording and storage of configuration data; i.e. details of configuration items and their versions and variants, list of changes made to configuration items, and recording and storage of proposed and processed changes. Configuration auditing checks that the change

activities are performed as defined in the configuration management standards and procedures.

3.5.4 The AMES model

The AMES model for software maintenance (see Figure 10) was used and refined during the first two case studies presented in this thesis. It is primarily intended for the software maintenance phase (Boldyreff et al. 1994), but it was applied to changes during the development phase in the first case study presented here.

Although the AMES model is mainly intended for use in the maintenance phase of the software life cycle, it is very relevant for the study presented in this thesis, since it was used as a starting point for the study of the change management. At the beginning of the AMES project, the goal was to focus solely on the maintenance process, but as the work progressed, the focus shifted more to managing change in all phases of the software life cycle.

The model has three main levels: strategic, operational and service level (Hather et al. 1995).

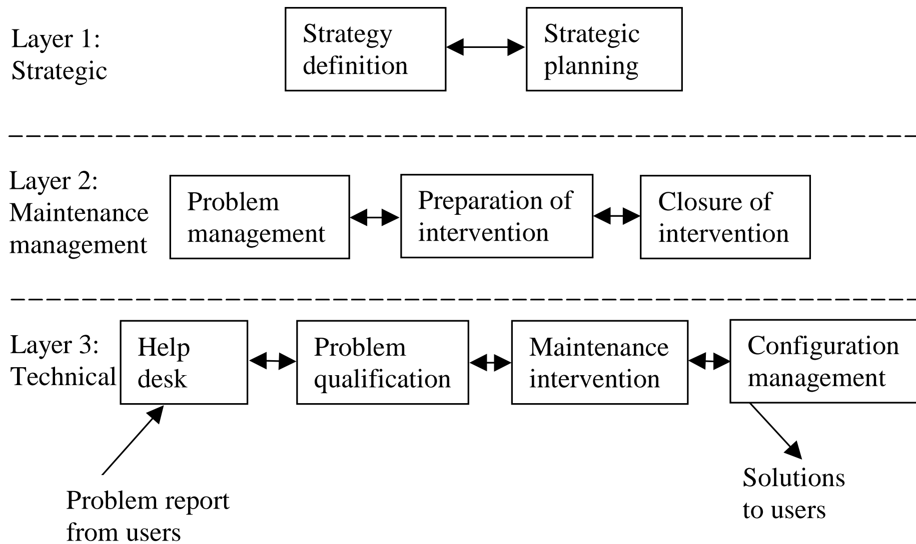


Figure 10. The AMES process model.

The AMES model supports the maintenance processes of a company. The model defines three layers: strategic, management and technical layer. The strategic layer takes care of decisions on planning the future of the product and how the customer or user relationship will be taken care of. The activities of the strategic layer include marketing, budget allocation, training and process improvement. The management layer plans, organises and controls the actions and people who provide the change service. The management layer includes activities related to progress tracking and planning, implementation decisions, problem management and initiating and closing the change. The technical layer carries out the changes. The activities of the technical level are similar to those presented by the V-like software change model, i.e. problem understanding, localisation, testing, change implementation and documentation.

3.5.5 Spiral-like change management process

When two of the three case studies presented in this thesis were completed, our view of the change management process was as presented in Figure 11 (Mäkäräinen 1996).

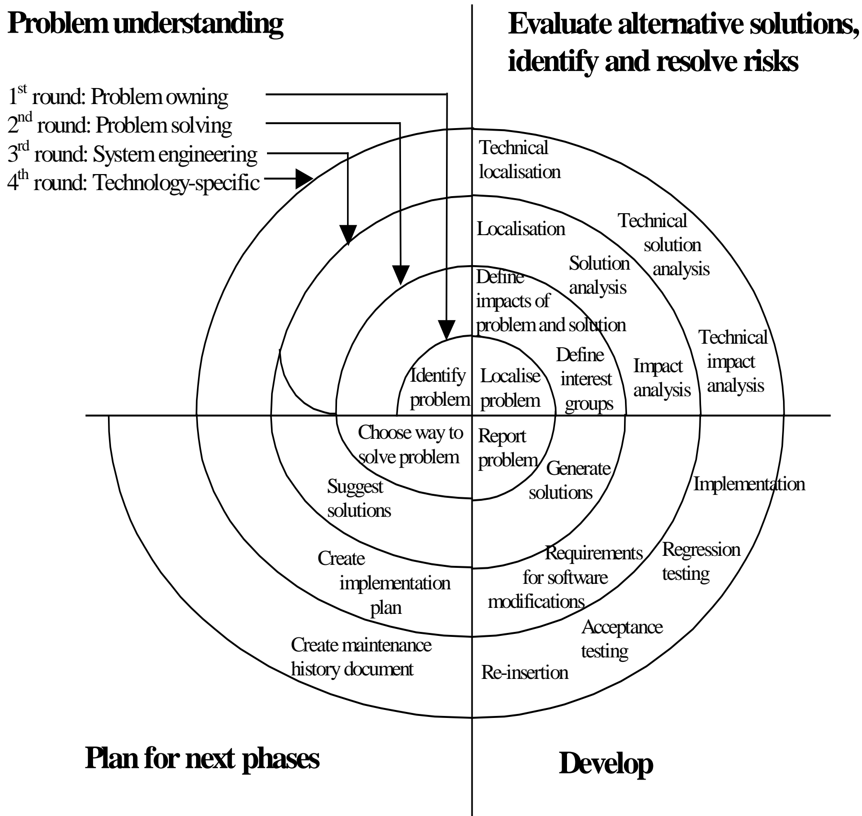


Figure 11. The spiral-like change management process.

The spiral-like process divides the change management process into four cycles, in which the same main tasks are performed by each cycle, but the viewpoint is different in each cycle. The execution of the process starts from the innermost cycle. The first cycle is performed by the founder, or "owner", of the problem. The problem at this point can be either:

- A request for something new, e.g. a new feature or service
- A problem in an existing product, e.g. an error situation in a product

As a result of the first cycle, the owner of the problem decides if the problem needs to be taken care of, and how it should be taken care of. The second round

of the spiral is optional. It is executed if the problem needs to be examined from a non-technical viewpoint. If the technical solution is known and clear after the first cycle, the second round can be skipped and the execution can be continued in the third round.

The third round of the process examines the modification from the system point of view, and makes an implementation plan for the last round. At this point, the affected parts of the system and the requirements for the modification task are forwarded to the fourth and final round of the process. The final round generates, implements and verifies the technical solution planned during the third cycle. It also closes the modification action by delivering the result and documenting the actions and observations made.

The process is generic from the organisational point of view, i.e. it has to be instantiated for the organisation using it. It is not life cycle dependent, but models the changes in both the development and maintenance phases. Different types of changes are not considered by the model.

The spiral model was derived in the AMES project, when the case studies showed that change management was relevant not only in the maintenance phase, but also in the development phase of the software life cycle. The spiral model does not treat software change as a post-delivery activity. Ince's change management model describes only the outer cycle of the spiral model and the last quarter of the system engineering cycle, i.e. the actual technical implementation of the change. The V-model was used in defining the outermost cycle of the spiral model. Olsen's change model treats all software development activities as changes. The spiral model only addresses the actions performed for changing existing pieces of work, not creating new artifacts.

3.6 Software change management – Technological dimension

Change management support has to involve all of the following facets: the groups involved with the change and change activity have to be co-ordinated and managed, the change must be supported by organisational change management process models, and technical support for change activities has to be pro-

vided (Michelis et al. 1997). Software change management tools and systems do not aim at reducing the rate of change, but they aim to reduce the overall costs of change management by facilitating the speed with which changes can be processed (Jones 1996).

The techniques used for supporting change management can be categorised into generic software engineering techniques and change management specific techniques (Mäkäräinen 1996). Generic software engineering techniques have been designed or are used in general for software development, but are vital in software change management, as well. These techniques include, for example, software configuration management environments (Ince 1994) and basic development tools, such as compilers, debuggers and editors.

However, special techniques for supporting software change management exist. These include, for example:

- Impact analysis techniques for analysing and modelling the impacts of the modification in the system. These techniques are useful, for example, in project schedule estimation, consistency checking and risk analysis (Arnold 1993, Arnold & Shawn 1996, Queille et al. 1994).

- Change request tracking to support management of change requests. The change requests are triggers for change. Their purpose is to (1) express a need for a change, and (2) to document change activities by providing change histories for individual changes and the software entities. The benefits of change request tracking systems usually are in documenting and communicating the changes.

- Reverse engineering techniques for deriving higher level descriptions from lower level presentations to help in understanding the software and improving its quality by the terms of understandability and consistency by updating outdated documentation (Chikofsky & Cross 1990, Sneed 1995, McClure 1989). Examples of such techniques are tools generating graphical design descriptions from source code, for example the ReverseNICE tool by Intecs Sistemi for generating HOOD descriptions from Ada source code.

- Regression testing for assuring that the modification has not created undesirable side effects in the system. Regression testing tools usually repeat old test cases and compare the new test results with the old ones in order to find out deviations.

One of the greatest challenges of change management is to keep the system parts and several abstraction levels of the system consistent with each others. This can be supported by an integrated software development and maintenance environment, where the tools used for creating and managing the software systems are able to communicate with each other and can share common parts (Cutillo et al. 1996).

Figure 12 presents an example of an integrated software change management environment, where the tools used for software modifications and development are able to communicate with each other and share knowledge. The environment was developed in the AMES project (AMES 1993). The tool environment presented here was constructed to support change activities defined by the V model for software change. The technical dimension of change management is therefore defined by the selected process model. The change process (adapted from Harjani & Queille 1992) is presented in the upper part of the diagram and the individual tools used in change management are linked with the process via a process support tool. In this example, change request tracking is performed by the process support tool. The interoperability service provides a link between the tools. All software related items are stored in one archive, which is used by all the tools through software configuration management. Links between semantically related software parts, different abstraction levels, composition structures, etc. are stored in a traceability database.

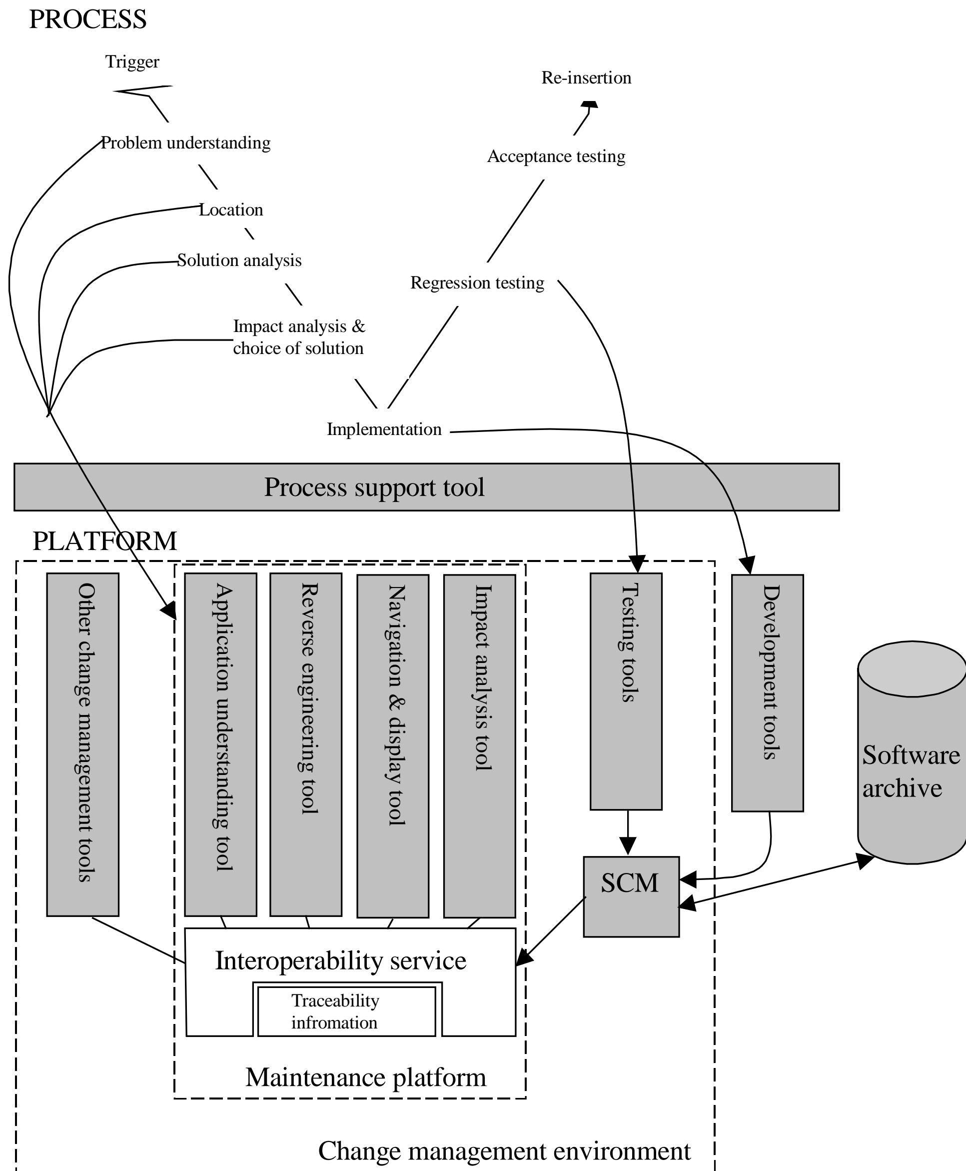


Figure 12. An example of an integrated change management environment (Mäkäräinen 1996).

An integrated change management environment was developed in the AMES project using the requirements elicited from the case studies (the case studies

presented in appendices A and B were done within the context of the AMES project). The project delivered a set of methods and tools for managing software changes.

3.7 Summary

This chapter presented the results of the literature study focusing on the research related to software change management and its relationships to other related concepts. The typical change management problems presented in the literature were discussed. The special features of the domain of embedded software which create special needs and requirements for software change management were presented. The process and technology dimensions for improving software change management were discussed.

4 Analysis of the state of the practice

4.1 Overview

In order to get an understanding of the state of the practice viewpoint to software change management, three case studies were conducted to study the change management processes in companies developing embedded software. The goal was to characterise, describe and analyse the change management processes identified in the companies, and learn what the problems in change management in software development and maintenance work in practice are. As a result, descriptions of the current status of change management in companies, and a list of problems identified and initial requirements for improvement were derived. The problem list was derived directly from the interview notes, or from the discussions and brainstorming sessions arranged to analyse the interview findings. The problems and requirements were directly stated by the interviewees or attendees of the brainstorming sessions.

The analysis of problems in the organisations studied aims at complementing the general change management problems presented in the literature and discussed in chapter 3.3. The problems identified in the case studies are related to the generic problems discussed in chapter 3.3. This shows what kind of practical problems can be caused by the generic change management problems discussed in the literature.

4.2 Summary of case studies

The summaries of the data collected from the case studies are presented as case narratives in appendices A (case one), B (case two) and C (case three). This chapter provides a short summary of the commonalities and differences in the characteristics related to change management in the three organisations studied. All the organisations produce products with embedded software. The two first organisations are small companies, and the third one is a large, geographically distributed corporation.

The context of software change management is very different in each organisation. In the first case study, the change management is relevant only during the initial development of the product, since the maintenance of the software is not possible after delivery. The second case study represents the 'traditional' view to software change management, where the main change effort is done after the delivery of the product. The third organisation studied uses an iterative, evolutive approach to software development. The iteration cycles deliver a new version of the product by modifying the version developed in the previous iteration cycle.

At the beginning of the research work, the initial assumption was that the focus would be on supporting the maintenance phase of the software life cycle. As the case studies progressed, it became clear that the change activities were not only related to the post-delivery related maintenance phase, but that the changes processed during the initial development also required support.

The enumeration of the requirements and problems was made only for the purpose of referencing them later. The numbering logic in each case is slightly different. In the first case study, the numbers are used to structure requirements. For example, requirement RQ-100 has sub-requirements, which are numbered RQ-110, RQ-120, etc. If the sub-requirement RQ-110 also has sub-requirements, they are numbered RQ-111, RQ-112, etc. In the second case study, the numbers do not express the structure of the requirements. The numbering of the requirements is ascending, and the structure of the requirements is expressed with chapter headings.

4.2.1 Case one

The first organisation studied develops unique space instruments. The development time is rather long, and the maintenance phase does not exist at all, since the instrument software cannot be changed after the take-off. Since the instruments are unique, they are specified and built from scratch in each development project. Therefore, during the initial phases of the project the changes are mostly requirement-level changes, since the hardware and software environment evolves all the time and the specifications get more detailed as knowledge cumulates and the optimal implementation solutions are found. As the environment gets more stable and the project proceeds to the testing phases, the nature

of the changes shifts to corrective work. The test coverage and the reliability of the corrections have to be high, since corrections are impossible after the launch of the instrument.

Two change process layers can be identified:

- Contractor level changes.
- Subcontractor level changes.

The procedures for managing changes at the contractor level have been agreed between the subcontractors and the main contractor. These changes are recorded on problem report forms, and the forms with the accompanying change information are filed. The decisions are managed by the board, which includes participants from the contractor and subcontractors. This procedure has been adapted from the ESA process guidelines (ESA 1991).

The procedures for managing changes at the subcontractor level are informal. These changes include all changes which are managed internally by the subcontractor, including errors found in the testing phases and modifications on the internal interfaces of the device constructed by the subcontractor. Before the first delivery of the product, even the interfaces between devices constructed by separate subcontractors are managed at the subcontractor level. After the first delivery and the launch, these are managed at the contractor level.

A summary of the change management problems identified in the first case study is presented in Table 2. The problems are explained in more detail in the appendix. The last column of Table 2 relates the practical problems identified in the case study to the generic change management problems discussed in the literature and discussed in chapter 3.3.

Table 2. Problems identified in case one.

Problem	**Description**	**Problem identified through the literature study**
Problem 1.	Frequent changes to specifications throughout the development time have made the documentation inconsistent.	Often the modification has to be done as quickly as possible. Few methods for supporting software modification. Many different tools are used and each tool manages only a subset of related information.
Problem 2.	Inadequate version control procedures.	Few methods or tools for supporting software modification.
Problem 3.	Inadequate testing procedures for verifying changes.	Few methods or tools for supporting software modification Verification of a change is complicated.
Problem 4.	Source code is very hard to understand.	Many software systems are old. The software is not designed for easy maintenance or modification. Maintainability is poor.
Problem 5.	Deficiencies in communicating requirement changes between development groups.	Few methods and tools for supporting software modification. Verification of a change is complicated.

The problems in keeping the documentation consistent with the constant flow of changes are related to the lack of the methods and tools for keeping the system documentation consistent throughout the long change history. When there is pressure for making the change as soon as possible and the consistency checking has to be done manually, the risk of forgetting to update part of the documenta-

tion is considerable. The automatization of consistency checking is difficult partly because the tools for software development and modification are separated and cannot share information with each other.

The version control and testing procedures and tool support were inadequate in the case organisation. Testing problems also relate to the generic problem of verifying a change in a complicated environment. This also caused difficulties in communicating changes between development groups, since it was difficult to verify which parts of the software were affected by the change.

Since the software is old, and it has not been designed or implemented to support modifications, the source code of the software system is often difficult to understand.

The requirements for improving the change management processes are listed in Table 3. The detailed list of requirements can also be found in the appendix.

Table 3. Requirements for an improved change management environment in case study one.

<table>
<tr><th>Requirement</th><th colspan="2">Description</th></tr>
<tr><td rowspan="6">RQ-100</td><td colspan="2">Automated configuration management tool for:</td></tr>
<tr><td>RQ-110</td><td>Managing all product materials</td></tr>
<tr><td>RQ-120</td><td>Ensuring file integrity after transfers and compressions</td></tr>
<tr><td>RQ-130</td><td>Compiling a configuration status list.</td></tr>
<tr><td>RQ-140</td><td>Compiling a configuration item data list.</td></tr>
<tr><td>RQ-150</td><td>Compiling a change request list.</td></tr>
<tr><td>RQ-200</td><td colspan="2">Automated links between the source code and the documentation for keeping the documentation consistent with the source code.</td></tr>
</table>

Table 3. Continues.

Requirement	Description	
RQ-300	Regression testing support for:	
	RQ-310	Managing test data versions
	RQ-320	Replaying testing sessions
	RQ-330	Interfacing with emulators and simulators
	RQ-340	Integration and unit testing
	RQ-350	Test case description
	RQ-360	Setting checking values for test results
	RQ-371	Static analysis
	RQ-372	Dynamic analysis
	RQ-373	Test coverage measurement
	RQ-374	Test result reporting
RQ-400	Impact analysis support for identifying how:	
	RQ-410	Changes made at a higher abstraction level affect components at a lower abstraction level
	RQ-420	Changes made at a lower abstraction level affect components at a higher abstraction level
	RQ-430	Changes made at any level affect components at the same abstraction level
	RQ-440	Changes made at any level affect test data
	RQ-441	The old test data should be updated
	RQ-442	To define which regression test cases to run after the change
	RQ-450	Changes made at any level affect the user documentation

Table 3. Continues.

Requirement	**Description**	
RQ-500	Process management support for:	
	RQ-510	Activity description
	RQ-520	Process tracking and control
	RQ-521	Problem report management
	RQ-522	Transmission of tasks and documents between people
	RQ-523	Reporting on the state of the tasks and the documents
	RQ-530	Instantiating company level methods and procedures for projects
RQ-600	Program understanding support for:	
	RQ-610	Managing multitasking architectures
	RQ-620	Managing multilingual programs
	RQ-630	Generating data flow and control diagrams
	RQ-640	Generating cross-reference information
	RQ-650	Extracting information about module interconnections
RQ-700	Requirements for the technical platform:	
	RQ-710	Sun SPARC with Solaris 1 or Solaris 2
	RQ-720	Motif or OpenLook interfaces
	RQ-730	Support for the C language
	RQ-740	Ada support is an advantage, but not a compulsory requirement

4.2.2 Case two

The second case study organisation develops consumer electronics. Critical factors from the change management viewpoint are the large amount of products distributed geographically, and the long maintenance phase of these products. The main change management effort is done after the release of the product, when the product software has to be updated and modified. Another typical change management situation is developing a new generation of products based on the experiences and improvement ideas derived from the old products. Reuse occurs mostly on the product concept level, since the hardware is usually unique for the product family, resulting in difficulties in implementation level reuse.

Commonly agreed change management procedures during the initial development do not exist. Official procedures exist for reporting error situations in operating devices located in the field, but these procedures are not always followed. Since the service personnel and external customers, who report the error situations in operating devices, know the person responsible for the software by name, they usually contact him to report the problem situations. The problem report is then filed by the person responsible for the software, instead of the original initiator of the error situation, as stated by the official procedure. All error situations are not reported. The person responsible for the software decides how the reported error situations are processed further.

Commonly agreed procedures for managing other types of changes or change requests do not exist. Their processing is left to individuals and is ad hoc. During the initial development, the changes are processed and managed by the software designers. After the device is delivered to the field, the person responsible for the software manages the changes as he sees fit.

A summary of the change management problems identified in the second case study is presented in Table 4. The problems are explained in more detail in the appendix. The problems identified through the literature study (listed in chapter 3.3) relating to the specific problems identified in the case study are presented in the last column of the table.

Table 4. Problems identified in case two.

Problem	Description	Problem identified through the literature study
Problem 1.	Testing is time-consuming and test coverage is unknown.	Few methods or tools for supporting software modification. Verification of a change is complicated.
Problem 2.	Impact analysis is difficult.	The software is not designed for easy maintenance.
Problem 3.	Error location is difficult.	The software is not designed for easy maintenance. The maintainability of the software is poor.
Problem 4.	Service personnel have difficulties in analysing product problems.	Few methods or tools for supporting change management.
Problem 5.	Inconsistent documentation.	The original requirements have been lost.
Problem 6.	Unstructured and unclear change process.	Few methods or tools for supporting modification.
Problem 7.	Version management of development tools.	Few methods or tools for supporting modification.

Several of the problems identified in the second case study were related to the lack of method or tool support for software maintenance. The overall change processes were unstructured and unclear, and the specific change tasks, namely version management, locating the problem by service personnel and testing, were not supported. The software documentation was inconsistent, which resulted in losing the original requirements for the system. The maintainability was neither designed nor built for the system, resulting in difficulties in locating errors in the system and estimating the impacts of the change.

The requirements for improving the change management processes are listed in Table 5. The detailed requirement list can also be found in the appendix.

Table 5. Requirements for an improved change management environment in case study two.

Requirement	**Description**
RQ-1	Change processes should be modelled
RQ-2	Change processes should be quantitatively monitored.
RQ-3	Change processes should be better guided and instructed.
RQ-4	Parallel projects should be able to share information.
RQ-5	Support for distributed project management.
Configuration and version management	
RQ-6	Any version of any module can be used
RQ-7	Support for configuration item list
RQ-8	Management of binary and map files
RQ-9	Developer control over the configuration item list.
RQ-10	Parallel product versions should be avoided.
RQ-11	Separation of modification tasks during implementation.
RQ-12	Support for change logs.
Reverse engineering	

Table 5. Continues.

RQ-13	Extraction of state transition diagrams.
Requirement	**Description**
RQ-14	Links between documents to support automated consistency checking and updates.
RQ-15	Graphical navigation between the state transition diagrams and the source code.
Modification request management	
RQ-16	Support for collecting error codes.
RQ-17	Support for managing and documenting error situations.
RQ-18	Modification requests should be managed by the SCM tool.
RQ-19	Support for monitoring modification requests.
Regression testing	
RQ-20	Support for static testing of the C code
RQ-21	Support for dynamic testing of the C code
RQ-22	Determining test coverage
Impact analysis	
RQ-23	Support for analysing how source code modifications affect documentation.
RQ-24	Support for analysing how source code modification affects other parts of the code.
RQ-25	Support for analysing how source code modification affects test data.
RQ-26	Support for defining which regression test cases are necessary after the modification.

4.2.3 Case three

The third case study organisation was studied separately in order to transfer the results and learning from the AMES project to the organisation. The focus was set right from the beginning on the change processes taking place during iterative development cycles. As the case study summary has shown, separating 'development' and 'maintenance' phases from iterative product development cycles is not practical in this organisation.

The third organisation has a set of product families, which are developed in an evolutive manner. Each development project gets an old baseline and a set of new requirements and changes as an input for the project, and develops a new version of the product accordingly. The software development work is actually change management, not building software from scratch. The post-delivery projects only perform emergency corrections and user support, the larger enhancements are made in evolutive development cycles. Therefore, separating changes made 'before' and 'after' delivery is not practical.

Commonly agreed, official procedures exist and are followed for managing errors initiated by testing activities and in review sessions. The processes have been defined, and tool and method support for handling these types of changes exist.

Requirement additions and modifications as well as improvement proposals are managed by individuals on an ad hoc basis. In particular, managing requirement level changes between different technology groups developing the same product causes lots of problems because the processes for managing requirement level changes have not been defined.

A summary of the change management problems identified in the first case study is presented in Table 6. The improvement requirements were not derived in this case study, since the purpose was only to analyse the change processes, not to build a new change management environment, and because the two first case studies showed that separating problems from requirements did not provide added value, since the problems and requirements were tightly related. Since the requirements are not listed in this case study, the problems include more problems related to current change management methods and environments. There-

fore, the text 'Problem in method used in the company' was used in the column containing the references to the generic change management problems discussed in chapter 3.3, to state that the problem is not caused by any of the problems identified in the literature study, but is a deficiency in a specific method or tool used in the company.

Table 6. Problems identified in case three.

Problem	**Description**	**Problem identified through literature study**
Problem 1.	Reviews reveal trivial, cosmetic defects, which will not cause errors or extra work in later phases.	*Problem in method used in the company*
Problem 2.	Release tests reveal defects which are not real defects.	The design knowledge and the rationale behind the design decisions have disappeared. The original requirements have been lost.
Problem 3.	How to minimise requirement level changes?	Many software systems are old. Problems and change needs tend to accumulate. The software is not designed for easy maintenance or modification.
Problem 4.	How to identify code or system decay?	Many software systems are old. Few methods or tools for supporting software modification.
Problem 5.	How to get relevant change information from other groups?	Often the modification has to be done as quickly as possible. Few methods or tools for supporting software modification.

Table 6. Continues.

Problem	**Description**	**Related generic problem**
Problem 6.	How to inform other technology groups about software changes which affect their work?	Few methods or tools for supporting software modification. The verification of a change is complicated.
Problem 7.	There are no defined channels or managed procedures for managing internal improvement proposals.	Few methods or tools for supporting software modification.
Problem 8.	Traceability information between a change request document and the modified document is often missing.	Few methods or tools for supporting software modification.
Problem 9.	The original reason for the modification is usually not recorded.	The design knowledge and the rationale behind the design decisions have disappeared.
Problem 10.	Effort spent for each modification is not recorded.	*Problem in the method used in the company*
Problem 11.	Locating the parts to be modified is time consuming and error-prone.	The software is not designed for easy maintenance. Verification of a change is complicated.
Problem 12.	Regression testing planning could be more efficient.	Few methods or tools for supporting software modification.
Problem 13.	Review tool supports document-based review, when feature-based reviews would be more practical.	*Problem in the method used in the company*
Problem 14.	All links between error reports, files, etc. are manual.	Few tools for supporting software modification.

Table 6. Continues.

Problem	Description	Related generic problem
Problem 15.	It is difficult to find information and compile summaries from text document based error logs.	Few tools for supporting software modification.
Problem 16.	The current SCM procedures prevent unit testing before the module has already been pretested.	*Problem in the method used in the company*
Problem 17.	No systematic analysis of change data for learning purposes is done.	Few methods for supporting software modification

Method and tool support were required for several change management tasks, namely for identifying code and system decay, communicating change information between development and project groups, managing internal improvement proposals, supporting traceability, planning regression testing and analysing change data for learning purposes.

The loss of original requirements and the design rationale behind the design decisions made caused false error reports. Since the release test designers did not know the requirements and the rationale behind them, they interpreted the behaviour of the system wrongly, and reported an error even if the system behaved as it was designed to behave.

The requirement-level changes were frequent in the projects, since the products under work were rather old. The implementation of requirement level changes was difficult, since the software was not designed for easy modication. Therefore, a change in one requirement easily led to modifications in others. The long lifetime of the system and the frequent changes also resulted in code and system decay.

4.3 Summary

The three case studies summarised in this chapter and presented in appendices A to C provide a practical viewpoint to the state of the practice software change management. The cases all have very different needs and constraints for change management. Some of the characteristics of the cases are listed in Table 7.

Table 7. Characterisation of background cases.

	Case 1	**Case 2**	**Case 3**
Degree of re-use	Software built from scratch	Some design level reuse, but mostly from scratch	High reuse level. Iterative, evolutionary development
Separation of development and maintenance	Maintenance not possible, only development phase exists	Short and clear development, long maintenance extending through product lifetime	New software releases as results of new development projects
Main focus of change management activity	Development	Maintenance	Development cycles
Number of products to be managed after delivery	Only one end product and product version, which cannot be modified after delivery	Large amount of products, several product versions	Large amount of products and product versions
Late integration	Critical, different devices cannot be integrated until in late phases	Low priority for change management, since during maintenance the hardware environment stays stable	High priority during first development cycle, when the hardware is concurrently designed with software. In later cycles, low priority

The level of software reuse varied a lot between the cases. The first organisation studied could not reuse anything, but had to start building the system from scratch. The second case organisation reused concept level designs, but did not reuse implementation solutions. The reuse level was highest in the third case study, where the development cycles were based on an existing software and product platform.

The products developed by each company also had very different product life cycles. The first case organisation designed products which could not be maintained after the release. The second company had a relatively short waterfall type development phase, after which the product underwent a long maintenance phase during its operation. The third company released new versions of the same product as results of successive development cycles. These development cycles took place throughout the operational life of the product. These facts lead to differences in how the companies emphasised software change management in different life cycle phases. The first company needed change management support only in the development phase, the second one mainly in the maintenance phase, and the third one concentrated on supporting evolutive software development cycles.

One complicating factor for change management proved to be the amount of products which share common software parts. The amount of products also varied considerably between cases. The first company studied delivered only one end product, while the second and third companies released a large amount of products with common software. In the second and third company, this was further complicated by the fact that the products in operation included different component versions, both hardware and software, because of component changes made over time.

Some differences could also be recognised in the importance of late integration of software components with other hardware and software components. In the first company, this was a very critical phase, since all devices were available at a very late stage of the development, and the success of the integration was mandatory because the faults could not be corrected after the release. The integration was also of high importance in the third case in the first development cycle, but its importance decreased in the later development cycles. The inte-

gration was not critical in the second case, since the hardware environment stays rather stable during the operation of the product.

At the beginning, the focus of the research was on post-delivery changes, but as the case studies progressed, they directed the research focus towards treating software change management as a life cycle independent activity. From the process and technology viewpoint, the case studies did not reveal any differences in change management before and after the first software delivery.

5 Software change management problems

5.1 Overview

This chapter presents a classification of the change management problems derived from the literature study and the case studies presented in the previous chapter. In addition, a summary of the problems and improvement requirements identified in the case studies related to each problem class is given. The summary also addresses the specific problems in the change management of embedded software development.

5.2 Classification of problems and improvement requirements in change management processes

The literature study and the case studies revealed problem areas and improvement requirements in the change management practices. The following six main problem groups can be derived from the problems identified in the three case studies:

- Effectiveness problems
- Communication problems
- Analysis and location problems
- Traceability problems
- Decision-making problems
- Tool-related problems

The following paragraphs explain each problem category in detail and give examples of the problems related to each problem class. The problems related to

the special features of developing embedded software listed in chapter 3.4. are examined separately for each class.

The purpose of the problem classification is to provide a framework for analysing the problems related to change management. The framework supports the change management process analysis task by providing a structure which can be used for identifying the change management problems in the process under analysis. The classification will be used in chapter 7 for analysing and structuring the problems found in change management, and later in chapter 8 to evaluate how the new change management procedures introduced improved the change management processes.

5.2.1 Effectiveness problems

This category includes the problems related to the effectiveness of identifying needs for change and the effectiveness of the change activities. Effectiveness is defined here as an ability to achieve the desired results, i.e. to perform change management activities so that all relevant changes and change needs will be processed within constraints which are practical and reasonable from the project point of view.

Identifying changes effectively means that all relevant changes and change needs are found and managed as early as possible. The problem covers not only the problem of managing all relevant change requests, but also the problem of avoiding unnecessary change requests, which do not require software changes, to decrease the analysis work needed to process the identified change requests. These change requests include misunderstandings, duplicate change requests, requests requiring user guidance instead of software changes, etc. (see Stark et al. 1994).

The most typical problems and improvement requirements related to the effectiveness of identifying the change needs in time were related to the effectiveness of reviews, inspections and testing. For example, the interviewees in case three felt that reviews were not effective because not enough time was used by the participants to prepare for the review, the work load of the people who possessed the best knowledge about the subject reviewed was already excessive, and poor review or inspection methods were used (see problems 1 and 2 in case

three). The problems in testing effectiveness usually arose from inadequate planning and failure to achieve a sufficient enough test coverage, lack of support for dynamic testing (i.e. repeating test cases, managing test cases and test results, etc.) and difficulty of integration testing before the hardware environment was ready (see problem 3, RQ-300 and its sub-requirements in case one, problem 1, RQ-20, RQ-21 and RQ-22 in case two, and problem 12 in case three).

The special features of the embedded software system presented in chapter 3.4. are reflected in the effectiveness area as presented in Table 8.

Table 8. Effectiveness problems related to special features of embedded systems.

Special feature of embedded systems	**Manifestation of effectiveness problems**	**Link to case material**
Concurrent system engineering	• Early integration testing is difficult, because hardware environment is not ready in early phases. • Hardware requirements are incomplete at the beginning of the project, which leads to requirement changes on software created by hardware	• See RQ-100, RQ-400 and RQ-500 in case one • See RQ-4, RQ-10 and RQ-11 in case two • See Problem 3, Problem 6, Problem 11 and Problem 16 in case three
Hardware-oriented programming and code optimisation	• Source code is hard to understand, which leads to difficulties in reviewing source code. • Not much tool support available for testing.	• See RQ-300 and RQ-600 in case one • See RQ-13, RQ-15, RQ-20, RQ-21 and RQ-22 in case two
High reliability demands	• High costs and practical limitations of after-release corrections require high effectiveness of pre-release change identification.	• See RQ-24, RQ-26 in case two

Concurrent system engineering has two major effects on the effectiveness of change management. Firstly, when the software and hardware parts are developed concurrently, the integration of software and hardware cannot be done in the early phases of the development project (see problem 5 in case one). This causes delays in finding errors and incompatibilities. Early testing phases have to be performed without the complete hardware environment. This creates delays in detecting inconsistencies or incompatibilities between the hardware and software, and delays the testing of the requirements which are affected by both hardware and software components, such as power consumption and execution times. Late error detection leads to a far greater amount of correction work than what would be needed if the error is identified and managed at an earlier stage (Humphrey 1989, Ince 1994). Secondly, changes in one component may lead to changes in the requirements for another component. This can create one additional source for software requirement changes (see problem 3 in case three).

Since embedded software is very hardware-oriented, the source code is relatively hard to understand (see RQ-600 in case one). Reviewing optimised hardware-oriented source code is very laborious and requires special knowledge from the reviewers. Also, the development environments for the processors used in final products seldom provide advanced support for testing (see problem 1 in case two).

Since the reliability demands for the delivered embedded products are usually high, the need for changing the software after release has to be minimised. The effectiveness of pre-release change identification has to be high in order to avoid after-release change pressure. (see problem 3 in case one)

5.2.2 Communication problems

This category includes the problems of informing all necessary parties about the change requests, changes and related items.

Communication problems are typical in situations where the software modules are shared by several development projects, and the products or product versions share parts of software or other software-related items, such as test cases

or user manual parts (see RQ-4 in case two). Communication problems also arise between technology groups when the change affects several technological parts of the product (see, for example, problems 5 and 6 in case three).

Communication problems usually have three components: “Who?”, “How?” and “What?”. The “Who?” problems dealt with the problem of identifying who should be informed about the change (see RQ-522 in case one, and RQ-16 in case two). Since the amounts of change requests handled in the time critical phases of the development cycles, such as late testing phases, are huge, restricting the distribution list only to the relevant persons is necessary to avoid overloading people. The “How?” problems concerned the processes related to how to distribute information about changes and change needs (see problems 5 and 6 in case three). The processes were very informal in the organisations studied, i.e. the change requests were received in personal conversations by phone or E-mail, and other involved people or projects were informed in regular meetings or in person-to-person communication. The “What?” problem was usually faced when common solutions for the “How?” problem had been defined, for example when common form templates for reporting errors were defined, and the relevant fields had to be identified (for example RQ-17 in case two).

The special features of embedded software system affect the communication requirements as presented in Table 9.

Table 9. Communication problems related to special features of embedded systems.

Special feature of embedded systems	**Effect on communication**	**Link to case material**
Concurrent system engineering	• Close interaction needed between technology groups throughout the development time. • Communication mechanisms have to be usable and comprehensible by all technology groups, not only SW people.	• See RQ-540, RQ-520, RQ-522 and RQ-700 in Case 1 • See RQ-4, RQ-5 and RQ-16 in Case 2 • See Problem 5, Problem 6 and Problem 9 in Case 3
Sharing software parts	• Distributing change information to all parties using modified components. • Distributed decision-making when modifications affect modules used by several parties.	• See RQ-522 • See RQ-10 in Case 2 • See Problem 6 in Case 3

Concurrent system engineering results in changes, which have effects on several product technology parts. Therefore, close interaction and communication of changes is required between the technology groups throughout the development time (for example, problems 5 and 6 in case three). The development groups usually operate as separate development groups or projects, who work in their own environments. In case one, the development groups were different subcontractors working with separate devices, which would be integrated together in the final product. In case three, the development groups were different technological groups within the company, some working with a specific software com-

ponent, and others with hardware units. The working practices, tools, and vocabularies between the groups are different.

When several products share common software components, communicating changes to all affected parties becomes crucial (see RQ-4 in case two). The impacts of the changes on all possible product environments have to be evaluated and considered in order to generate implementation alternatives and to provide adequate and precise information for implementation decisions. The changes have to be distributed and validated in different environments.

5.2.3 Analysis and location problems

One of the most time-consuming and error prone phases of the change process is problem understanding in the context of understanding the system and analysing which parts of the system will be affected by the change (Barros et al. 1995). This was also clearly stated by the software engineers in the organisations analysed in the case studies (see problem 3 in case two and problem 11 in case three). Repeating the problem in an indeterministic environment (as in case two), locating the problem in a complex multi-technology environment (as in case three), analysing the ripple effects of the various solution proposals and achieving an understanding of the complex system were among the most problematic change tasks identified. This category includes problems related to the analysis of the change and change request, and relating the change to the system to be modified.

The special features of embedded software systems are related to the analysis and location problems as presented in the Table 10.

Table 10. Analysis and location problems related to special features of embedded systems.

Special feature of embedded systems	Manifestation of analysis and location problems	Link to case material
Concurrent system engineering	• When incomplete, not properly tested HW and SW components are integrated, it is extremely difficult to find out the actual root causes for errors.	• See RQ-600 in case one • See Problem 3 and Problem 4 in case two
Hardware-oriented programming and code optimisation	• Hardware-oriented and optimised source code is difficult to understand. • Special technical features, such as multitasking architectures and non-deterministic behaviour, have to be dealt with.	• See RQ-400, RQ-600 in case one • See Problem 3, Problem 4, RQ-8 in case two • See Problem 4, Problem 11 in case three
Sharing software components	• The module can be used in several different HW and SW environments, and the effects of the modification has to be analysed in all of them.	• See RQ-640, RQ-650 in case one • See Problem 2, RQ-7 in case two • See Problem 11 and Problem 12 in case three
High reliability demands	• Unwanted impacts of the modification have to be predicted before release, because corrective actions may be impossible or very expensive.	• See Problem 3 in case one • See Problem 1 in case two

Locating the problems encountered in late testing phases is extremely difficult, because the complete testing environment with all hardware and software components is not available until rather late. The integrated system is very complex. Locating problems and analysing change impacts on highly optimised and hardware-oriented source code is challenging (see RQ-600 in case one and problem 2 in case three).

Embedded systems often have special technical characteristics, which cause problems in analysis and location of changes. Examples of such characteristics are indeterministic behaviour (see problem 3 in case two), which complicates the task of repeating the error situation and hinders the use of automated test result analysers, and multitasking architectures, which make analysis of the impacts of changes more complex.

Sharing software parts in several product environments complicates analysing the impacts of the change (see problem 2 in case two, and problem 12 in case three). The impact analysis of embedded software products must also be efficient, since unwanted impacts may create problems which could result in expensive or irremediable error situations afterwards.

5.2.4 Traceability problems

Traceability problems are related to establishing and maintaining traceability links between software artefacts and related items. The traceability links are used for maintaining the consistency of the different abstraction levels of the system, and supporting program understanding by providing information about the related items. Support for establishing and maintaining traceability links between abstraction levels and within abstraction levels, and aid for keeping the system consistent by using traceability information was sought by all case organisations (see Problem 1, RQ-200, RQ-410, RQ-420, RQ-430, RQ-440, RQ-450 in Case 1, RQ-14, RQ-23, RQ-24, RQ-25 in Case 2, and Problem 8 and Problem 9 in case three). Manual creation and especially manual maintenance of the traceability links was found to be difficult and error-prone (see Problem 14 in case three). Also, traceability from software items to design decisions and reasons behind design decisions and other process related information, such as modification records, was regarded as valuable (see RQ-12 in case two and Problem 8, Problem 9 and Problem 17 in case three).

The traceability issues are general in nature; they are not directly related to any specific feature of embedded software systems. Development environments for embedded software seldom provide any support for managing traceability issues, but neither do the general purpose software development environments. The only special feature of embedded systems that has to be dealt with is the traceability between the different technology components of a product.

5.2.5 Decision-making problems

The decision-making problems arise from the difficulty of defining how the decision-making responsibilities should be assigned to keep the change processes simple enough but still under control. They also include the problem of how to present all relevant information to decision-makers and how to pass the decision and the reasons behind the decision to the implementation and later phases of the development cycle The problems are closely related to the communication problems.

The most typical decision-making problems were encountered in situations where the responsibilities had not been clearly defined. For example, the customers directly called the software designer to request new features to be implemented, and the feature was implemented by the decision of the designer. This may lead to inconsistencies in project plans, software components, etc. (see RQ-510, RQ-520, RQ-521, RQ-522 in Case 1 and Problem 6, RQ-3 and RQ-5 in Case 2) Decision-making problems were also encountered when the responsibilities for implementation decisions depended on the criticality of the change request, i.e. the implementation of a minor request could be decided by the designer, but the major ones had to be handled by the project manager (see case three). In these cases the difficulty was in defining the criticality of the change request.

The decision making problems are very general in nature. The only special feature related to embedded systems is the added complexity of decision-making introduced by the involvement of the hardware development. When the change affects several technology components of the product, the decision-making process has to take into account the hardware development groups, as well. Also, sometimes the change request can be implemented by means of software or

hardware, and the decision making process has to evaluate which one is the most beneficial solution option from the product viewpoint.

5.2.6 Tool-related problems

Tool-related problems are problems related to the tools used in change management activities in the organisation. These problems can be formulated into requirements for improved tool support, either for new tools to be implemented or for new, improved revisions of the old tools. This has been done in the first two cases presented in Appendices A and B.

Tool-related problems are very company-specific, depending on the specific tools used in the company (for example, problems 13 and 15 in case three). Therefore, it is not reasonable to list the typical problems related to companies developing embedded software. Instead, the tool-related problems and requirements can be classified according to the tool functions, as is done in Table 11.

Table 11. Summary of change management tool requirements.

Tool function	Case one	Case two	Case three
Process support	RQ-510, RQ-520, RQ-521, RQ-522, RQ-523, RQ-530	RQ-1, RQ-2, RQ-3, RQ-4, RQ-5	Problem 7
Application understanding	RQ-610, RQ-620, RQ-630, RQ-640, RQ-650	RQ-15	
Reverse engineering	RQ-200	RQ-13, RQ-14	
Modification request management		RQ-16, RQ-17, RQ-18, RQ-19	Problem 15

Table 11. Continues.

Tool function	Case one	Case two	Case three
Impact analysis	Problem 2, RQ-410, RQ-420, RQ-430, RQ-440, RQ-442, RQ-450	RQ-23, RQ-24, RQ-25, RQ-26	
Regression testing	RQ-300, RQ-310, RQ-320, RQ-330, RQ-340, RQ-350, RQ-360, RQ-371, RQ-372, RQ-373, RQ-374	RQ-20, RQ-21, RQ-22	Problem 12
Software configuration management	RQ-100, RQ-110, RQ-120, RQ-130, RQ-140, RQ-150	Problem 7, RQ-6, RQ-7, RQ-8, RQ-9, RQ-10, RQ-11, RQ-12	Problem 16

In general, embedded software development projects seem to require support from process-related tools especially for the communication and information distribution activities between technology groups. The requirements for tools supporting individual tasks, such as configuration management or testing, are usually very simple requests for basic support, since the development environments available for embedded software systems are usually rather primitive.

The requirements for each functionality are briefly discussed in the following subsections.

Process support

Deficiencies in formalising and modelling the change processes were identified in all three case studies (see RQ-500 and its sub-requirements in case one, problem 6 in case two and problem 7 in case three). Since the change processes were not formalised, no tool support was available at the time of the studies.

Therefore, the requirements for tool support focused on the basic process support functions, such as process monitoring (e.g. RQ-523 in case one and RQ-2 in case two), tracking (e.g RQ-520 in case one) and process flow control (e.g. RQ-522 in case one).

Application understanding

All case studies revealed problems in understanding the target system (see, for example, RQ-600 in case one). Several reasons contributing to the difficulty of understanding the systems built in the case organisations were identified:

- Indeterministic nature of the system (see problem 3 in case two). The behaviour of the system cannot be predicted from the input values. This causes problems in analysing root causes and repeating cases.

- Hardware-oriented, optimised programming style (see problem 4 in case one). Understandability of the software is not always one of the most important quality criteria, and it has to be sacrificed because of optimisation issues.

- Modification may have ripple effects on several products through common software components (see problem 2 in case two). This complicates the analysis, since the software engineer should be able to track several product combinations and analyse the effect of the change in all of those.

- Locating the problem in a complex device with several hardware and software components is often troublesome (see problem 4 in case two, and problem 11 in case three).

The requirements for supporting application understanding dealt with providing visualisations of the source code (see RQ-630 in case one, and RQ-13 in case two) and providing information about relationships within the system (see RQ-640 and RQ-650 in case one). The requirements for application understanding tools deal with providing support for understanding source code, since it often is the only reliable material available to help the understanding of an application. The documentation may be out-of-date, or there may be no documentation available (see Problem 4 in case one and problem 5 in case two). Also, tool sup-

port for updating the documentation when the source code is modified was requested (for example, RQ-420 in case one and RQ-14 in case two)

Reverse engineering

The requirements for reverse engineering tools, i.e. tools which would extract higher level descriptions from lower level artefacts, were very much directed towards the purpose of supporting application understanding tasks and keeping the documentation synchronous with the source code (see the requirements in the previous chapter). In addition, the third case study identified the problem of code and system decay, which could be addressed by means of reverse engineering techniques (see problem 4 in case three).

Modification request management

Modification requests were collected in all the organisations studied. Three types of requirements for improved modification request management arose:

1. Not all types of modification requests were managed equally well (see problem 7 in case three).

2. The procedures for managing modification requests were not followed for some reason (see RQ-16 in case two, where the service personnel did not report all error codes to the software department, and problem 16 in case three, where all unit test errors were not reported by the software engineers).

3. Modification request management should be supported by adequate tools and databases (see RQ-521 in case one, RQ-17, RQ-18 and RQ-19 in case two, and problem 15 in case three).

Impact analysis

Impact analysis support was requested in the context of analysing the ripple effects of the modification in order to ensure the consistency of the system after the modification (see RQ-500 and its subrequirements in case one and RQ-23 to RQ-25 in case two). The ripple effect analysis can be automated only if the links between the system items are somehow known. For that reason, the traceability

links between the system components have to be managed somehow. Manual creation and maintenance of traceability links was regarded as unreliable (see problem 14 in case three).

Regression testing

Change management related requirements for testing support arose from two directions:

1. Testing activities should identify errors (i.e. inputs to the change management process) effectively and efficiently (for example, problem 2 in case three).

2. Regression testing of a change should be supported (see RQ-320 in case one, RQ-21 in case two and problem 12 in case three).

The special problem in testing support within the context of embedded software is that the development environments seldom provide advanced tool support for testing activities (Vierimaa et al. 1998), and the testing environments with simulators and emulators are complicated to build and maintain (see RQ-330 in case one).

Software configuration management

Cases one and two had simple software configuration and version management systems in use, but no defined or established practices for using them. Both organisations were in the process of establishing consistent practices, and the change management related configuration management requirements were very much related to this improvement initiative (see RQ-100 and its subrequirements in case one and RQ-6 to RQ-12 in case two).

The third case organisation had configuration management procedures in place, and its requirements for software configuration management tools related to change management were focused on the issue of integration of configuration management and change management tools (such as problem 8 in case three, where a link between a file managed by the configuration management tool and

a change request document stored in the change management system was requested).

Case two also had a problem of managing the versions of the development tools, such as compilers and debuggers (see problem 7 in case two). As the development tools evolve, the software parts developed using the older versions of the tools may become inconsistent with the new versions of the development tools. This creates problems when modifications to these software parts have to be made.

5.3 Summary

This chapter presented the classification and summary of change management related problems and requirements derived from the literature study and case studies presented in the previous chapters. The problem classes were characterised, and the special problems identified in the change management of embedded software that were identified through the case studies were summarised.

6 Generic change management process model

6.1 Overview

This chapter presents a generic change management process model, which illustrates our view of change management activities after performing the three case studies presented in the appendices. The chapter determines the scope of software change management processes as they are defined in this thesis. The process model has been derived from the analysis of related work and complemented with the analysis of the three case studies summarised in chapter 4.

6.2 Background

In order to respond to the change management problems and improvement requirements presented in the previous chapters, the change processes needing support have to be defined. Without knowing the processes and activities to be studied and analysed, it is impossible to plan and evaluate relevant improvement actions. The generic change management process model defines the exact focus of the analysis and improvement actions. The generic change management process model proposed here has been derived from the literature study and the case studies presented in the appendices. The process model is used to describe how change management processes are defined in this thesis.

The generic process model aims at modelling the types of change management processes which were recognised in the case studies. The process model has been derived from two sources. First, the change process models presented in the literature, such as (Ince 1994) and (Harjani & Queille 1992), were studied. Then, the common features and special characteristics of the change processes recognised in the organisations explored in the case studies were synthesised with the models presented in the literature. The goal was to get both practical and theoretical background for defining the change management processes and process types.

6.3 Layered change processes

The case studies showed that changes are processed in several process layers. The process models formulated during the first case study have two process layers: contractor level change process and subcontractor level process. These two layers were very clear, since the process standards and requirement used by the contractor were very different from the processes used internally by the subcontractor. Therefore, the changes visible to the contractor were processed using very different processes compared with the changes which could be managed internally at the subcontractor level. The spiral-like change management process generated after the second case study includes four process cycles, each including the same main phases, but executed by different roles. The factor defining the process level was the role of the person who executed the change activities at that level. Two change process layers, product and project layer, were identified in the third case study, although one of these was part of the requirements management process. These two layers were the clearest ones identified in the third case study, although there were indications of additional process layers.

The following two process layers can be identified in all case organisations:

1. *Product level changes.* This change process generates new requirements for the new products or new product releases. It also generates feature modification, addition and deletion requests for development projects. Examples of product level changes from the three case studies are presented in Table 12. These examples were identified in the case studies.

Table 12. Examples of product level changes.

	Example of product level change
Case one	Adding or modifying the instruments in the station during the development time by the contractor of the whole project.
Case two	Replacing components in operating product families with modern hardware components. The new hardware components require software to be changed accordingly.
Case three	Requirements for new features for new iterative development cycles.

2. *Project-level changes.* Project-level changes are initiated and managed internally by the project. Some project-level changes are generated because of product-level changes, for example when the product-level change process requests a feature modification initiated by an external customer during the development project. Project-level change requests may also generate product level changes, for example when a project generates requirements for other, either ongoing or future projects. Examples of project-level changes identified in the three background case studies are listed in Table 13.

Table 13. Examples of project-level changes.

	Example of project-level change
Case one	A timing error in flight software caused by a programming error and identified in the unit testing phase.
Case two	Re-writing part of the software reusing the behaviour and concept of the software embedded in the old product. This is done because the old software has deteriorated during the long maintenance phase, and its structure and understandability has suffered from patch changes.
Case three	A typographical error in a design document identified in a design document review.

The main difference between the two change process layers is that the decisions related to product-level changes are done outside an individual software project, and the project-level changes are managed internally by the project within the constraints stated for the project. When the product-level changes are forwarded to the software project for implementation, they introduce changes to the initial

requirements set for the project. On the other hand, the project should be prepared to manage the project-level changes within the project constraints.

In addition to these two change process layers, the case studies indicated the existence of additional change process layers. For example, the third case study revealed that only a subset of unit testing errors are processed according to the project-level error management process (See Problem 16 in Case 3). This indicates the existence of a "personal change process", which is followed for changes in which the person managing the change does not forward the change to the project-level change process. The change processes can also scale upwards from the product-level changes. For example, in the organisation studied in the third case study, the products were grouped into product families. One of the goals of the product family thinking was to share concepts and even components, both hardware and software, between products. Therefore, change management on a product family level is necessary. The product family thinking was, however, a novel idea in the organisation, and therefore the product family change management level was not yet clearly recognised.

The process levels can be identified in both the development and maintenance phase of the software life cycle. The levels are also independent of the change type (see Figure 18). Different types of changes can be identified on all process levels.

The relationships between the product and project-level change management process layers are described in Figure 13.

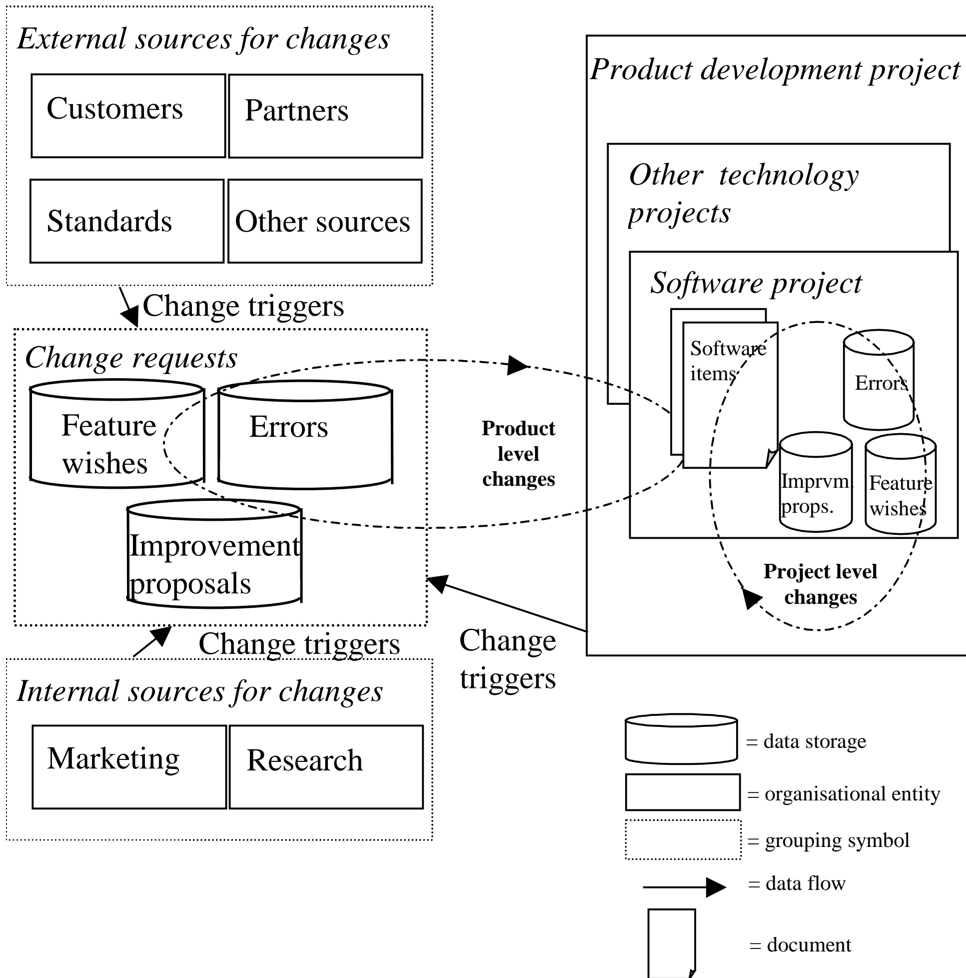


Figure 13. Relationships between the product change process and project change process.

The product and project level change processes were the most distinct and clear process levels identified in the case studies. These two change process layers are described in further detail in the following two subsections.

6.3.1 Product-level changes

When the nature of software development is evolutive, i.e. new products are based on the idea, components or a baseline of an old product, the product-level change process can be clearly distinguished from the project-level changes.

In the beginning of a new evolutive development cycle, the product-level changes are change needs or ideas related to the old product version, which are refined into requirements for the new development cycle. After the product-level change process has forwarded the change requests to the development project as the requirements for the new development cycle, the project will manage them using a requirement management process, since they constitute the requirement specification of the project. If the product-level change process feeds the ongoing project with new change requests, those changes are managed as change requests to the requirement specification baseline. These changes cause additions, modifications or deletions to the requirement specification baseline of the ongoing development cycle.

From the project point of view, the changes initiated by the product-level change process always have an impact on the assumptions on which the project bases its project plans. Therefore, the impacts to project schedules, resources and other plans must be analysed and updated.

The project may also feed change proposals to the product-level change process. The ongoing development cycle may generate change needs or ideas which it cannot carry out within the constraints of the ongoing development cycle. The project will then feed the change proposal to the product-level change process, which will evaluate if the change proposal will be forwarded to another ongoing development cycle, or to a new development cycle to be started.

The new evolutive development cycle does not have to be organised as a project, although that was the case in the two case studies (case studies B and C) where the product-level change process could be clearly separated from the project-level change process.

When software development is based on an evolutive software development cycles, the starting point of the project is:

- a stable baseline of an old product, and
- a list of changes to be implemented during this development cycle. The changes can be any one of the following:
    - new or modified features,
    - corrections,
    - adaptations to new environment, or
    - improvement proposals.

This means that the nature of the system development is actually adapting an old system version to a new release. The nature of the software development work is carrying out changes in order to deliver a new release of the product.

6.3.2 Project-level changes

Project-level changes are iterations within the development process. They are managed internally by the development unit, which usually is a development or maintenance project. Project-level changes are included in the project plans and they are managed within the constraints of the project. The project should be prepared to manage the project-level changes within the constraints set for the project.

In some cases, project-level changes may have effects on the assumptions used as the basis for the project plans. However, this is not caused by external pressure, but the change need is initiated internally by the project, i.e. because the knowledge of the subject area increases as the design work progresses, and the initial assumptions prove to be inaccurate or incorrect.

6.4 Generic change management process

The generic steps of the change process presented in Figure 14 can be identified in all change process instances.

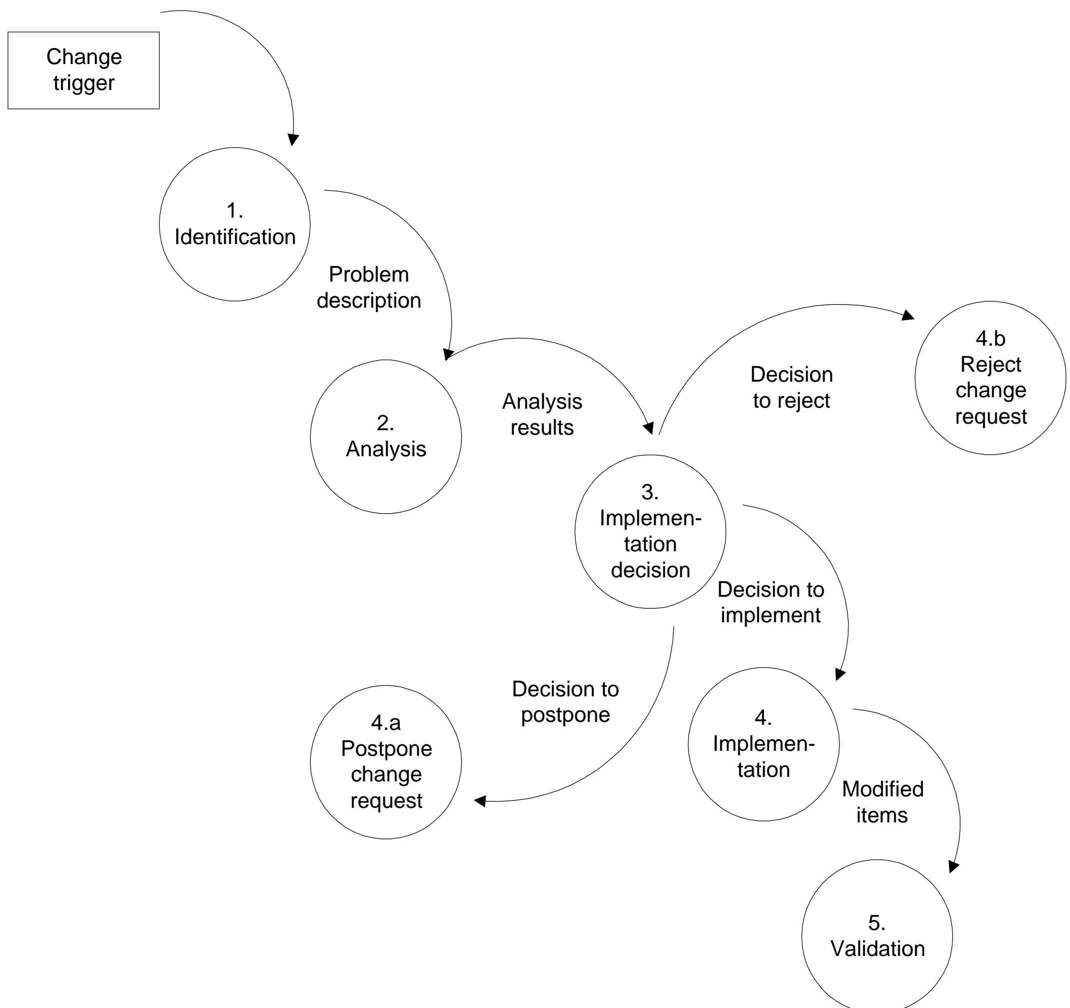


Figure 14. Generic change management process.

Four types of instances of the generic change processes were identified: trivial defect corrections, defect corrections, requirement level modifications and improvement proposals. The first two process types deal with corrective changes,

and the last two with enhancements. The change processes are briefly characterised as follows:

1. **Trivial defect correction.** This process is followed in the small, simple, straightforward corrections, which do not have to be separately analysed for the implementation decision, and can be corrected and verified directly after identification. The analysis and implementation decision steps are done simultaneously with identification. This group includes small errors found in unit testing, which do not affect other modules or product parts, and most defects found in reviews, since the implementation decision is done in the review meeting, and a separate step for the implementation decision is not needed.

2. **Defect correction.** This process is followed in managing most defects found in the integration and release testing phases, and after the product release. The unit testing errors which trigger change needs on other modules or technology parts follow this process as well. This process is also followed for managing the defects identified in reviews, where the implementation decision cannot be made in the review meeting because further clarification work is needed to support decision-making.

3. **Requirement level modification.** This process is followed when the project receives a request for a new feature, an existing requirement has to be modified or deleted, or other requirement level changes are needed. The requirement-level changes usually have effects on the estimates which have been used when the original project plans have been made. Therefore, risk analysis and revision of project plans are especially crucial when requirement-level modifications are analysed.

4. **Improvement proposals.** These modifications include improvement proposals and requests for actions which would prevent problems in the future. Examples of such actions are: restructuring the code, renaming, removing dead or duplicate code, etc. Triggers for preventive modifications usually come internally from the software project. Automated quality measures, such as complexity or defect density measures, can also be used in identifying software modules which may need preventive modification actions.

The process types can be roughly compared with the maintenance types defined by Swanson (Swanson 1976) as follows:

- Trivial defect correction and defect correction processes manage changes of the corrective type.
- Requirement level modification process manages :
    - corrective changes, when the correction is needed at the requirement specification level,
    - adaptive modifications, since changes in the environment have an effect on system requirements,
    - perfective changes, if perfective actions have effects on the requirement level.
- Preventive modification process corresponds to the preventive maintenance type added to Swanson's maintenance type classification later.

The change process types are characterised in following subchapters.

6.4.1 Trivial defect correction

Trivial defect corrections have typically only three main steps (Figure 15) :

1. Defect identification
2. Implementation
3. Validation

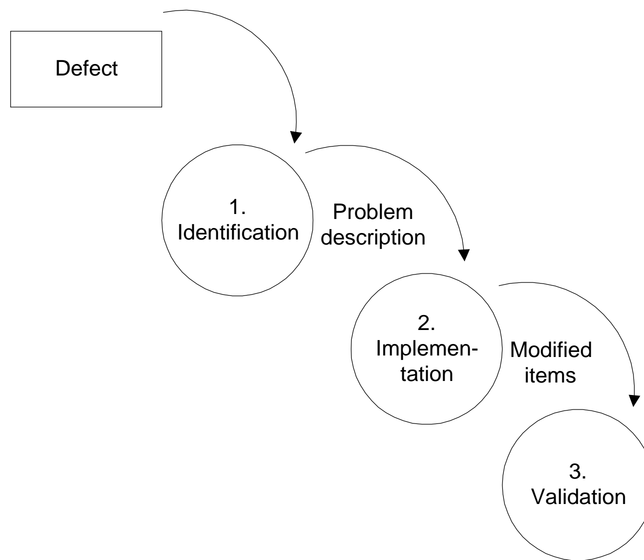


Figure 15. Trivial defect correction.

Since the defect is very straightforward in nature, it does not have to be separately analyzed, and a separate implementation decision step is not required for proceeding to the implementation phase. The defect analysis and implementation decision are made simultaneously with the identification phase. For example, in a review meeting the change requests are presented and discussed, and if the request is found to be irrelevant it is rejected already in the review meeting and the "defect" is never recorded at all (Figure 16). Respectively, if the defect identified in testing is small and trivial, the tester identifies, analyses, and makes the implementation decision in the testing session.

In the case of defects found in reviews, there is an additional life cycle for the review session. The combined review session, defect analysis and implementation decision process is presented in Figure 16. The process illustrated in the figure is a detailed description of step 1 of the process illustrated in Figure 15. After a review meeting the three-step trivial defect correction process illustrated in Figure 15 is followed for all defect recordings done in the review session.

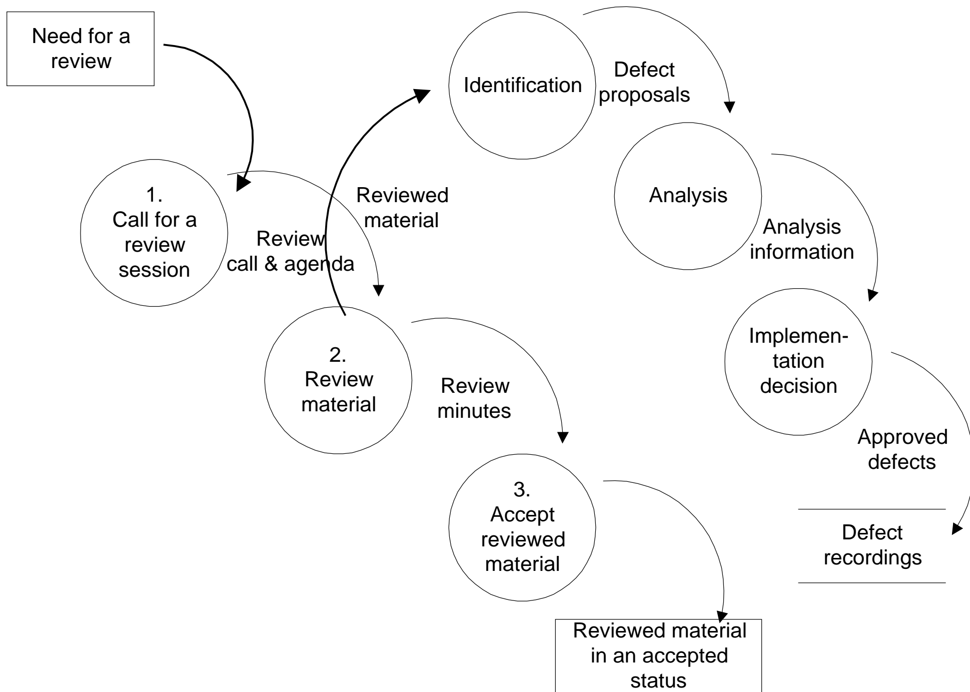


Figure 16. Defect identification in a review.

Examples of trivial defects are:

- Most defects found in reviews. The identification, analysis and implementation decision steps are already performed in the review meeting. The defects which are not simple and straightforward to manage (i.e. the implementation decision cannot be done in the review meeting) are handled using the defect correction process.

- Unit testing errors which affect only the modules tested and have no influence on other modules or product parts.

6.4.2 Defect correction

The defect correction process includes all defects whose correction method is not self-evident. The process follows the generic change process. Figure 17 explains the analysis phase of the defect correction process in more detail, since it is essential especially in the defect correction process. There are two important steps that have to be performed in the analysis phase:

1. Find out the location and impacts of the defect. This is crucial for defining if other groups have to be informed about the defect, and for finding out what the proper way to organise the correction is.

2. Analyse the causes of the defect. Some reported defects are not real defects, but rather misunderstandings of original requirements. Also, the root causes of several defects can be the same, allowing the defects to be grouped together and fixed simultaneously.

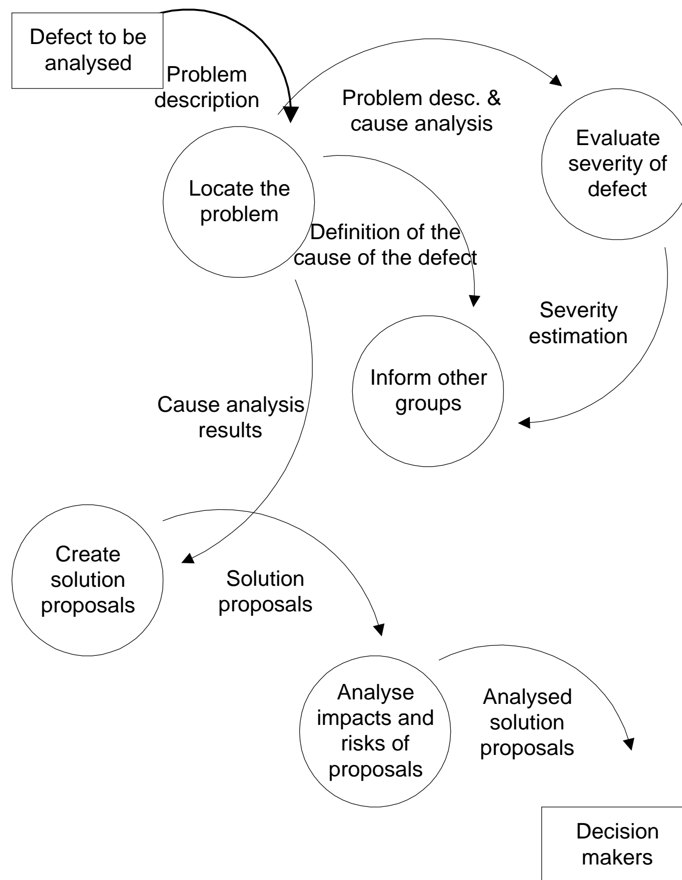


Figure 17. Analysis step of defect correction process.

Following defect features were used in the companies studied to define if the defect is complex:

- The defect has effects on other software components.
- The defect has effects on other product components.
- The defect has effects on the schedule or other plans in the project.

- There are several alternative solution proposals, which have to be evaluated and compared with each other in the implementation decision phase.

6.4.3 Requirement-level modification

Requirement-level modifications are modifications, deletions or additions to the requirement set of the project. The change process follows the generic change process (Figure 14), which has special features related to project re-planning especially in steps two, three and four of the process. The effects of requirement-level changes on project schedules, resources and other plans must always be analysed prior to the implementation decision, since the requirement-level changes usually change the estimates on which the plans are based.

Most requirement-level modifications intertwine with the product level change process, since the triggers for requirement modifications are usually received from outside the project, and the first identification, analysis and implementation decision steps may have already been done. They are repeated again in the project level with special emphasis on the analysis of the impacts of the modification to the software system and the subproject in question.

6.4.4 Improvement proposal

The improvement proposals usually initiate preventive modifications. Preventive modification actions follow the generic change process. As in requirement level modifications, the effect to project plans plays a very crucial role in the implementation decision. In effort estimation (made in the analysis phase), it is important also to estimate the risks and threats of not implementing the preventive modification. Sometimes preventive changes may actually save time and effort by eliminating problems in the future.

Some examples of improvement proposals are:

- Restructuring code to improve system structure and architecture.

- Re-documenting code, if the documentation is inconsistent with the code or does not exist.

- Reformatting or renaming code to improve its understandability and readability.

6.5 Relation of process levels and process types

The model adapts the change process with respect to two factors: change layers and change types. These two factors form a matrix of processes. An example of such a matrix is presented in Figure 18.

	Trivial defect correction	**Defect correction**	**Requirement level modification**	**Improvement proposal**
Project level	*Instance 1*	*Instance 2*	*Instance 3*	*Instance 4*
Product level	*Instance 5*	*Instance 6*	*Instance 7*	*Instance 8*

Figure 18. Process instantiations for two process levels.

In the example, the two most distinct change levels are presented. All change process types can be found in all change levels, e.g. improvement proposals are made in both project and product levels, and their management processes most probably differ between the levels.

6.6 Comparison to other models

The main similarities and differences of the proposed model and the models of change presented in the literature are shortly characterised in Table 14. The reference models are the models described in chapter 3.5.

Table 14. Comparison of the presented model against change models presented in the literature.

	Life cycle phase	**Adaptation to different change types**	**Change activities covered**
The model proposed by this thesis	Life cycle independent	Four distinct models for different types	Management of change activity
Olsen's change model (Olsen 1995)	Software development and maintenance	All changes follow the same process	All development activities are treated as changes
V-model (Harjani & Queille 1992*)*	Maintenance	One main process, variants for exceptional situations	Technical modification
Ince's model (Ince 1994)	Software configuration management	All changes follow the same process	Management of change activity
The AMES model (Hather et al 1995)	Maintenance	All changes follow the same model, different models can be used within the levels	Strategic, management and technical levels
Spiral model for change management (Mäkäräinen 1996)	Life cycle independent	All changes follow the same process	4 levels of technical modification

The change models are characterised in the table using the following dimensions:

- does the model view change management from a specific life cycle phase point of view,
- how does the model adapt to different change types, and
- what level of change activities does the model cover.

The model presented in this thesis treats change management as a life cycle independent activity, as does the spiral model, while the AMES model and the V-model concentrate on the post-delivery life cycle phase, i.e. maintenance. Ince's change model approaches change management from the configuration management point of view. Olsen's model treats all activities done in any software life cycle phase as changes.

All the models, except the one presented in this thesis, propose one change model for all types of changes, although some variation can be allowed in exceptional situations (Harjani & Queille 1992).

Our model covers the aspect of managing change activity and does not cover strategic or technical issues of change. The V-model covers only the technical activities for changing software, as does the spiral model for change management. The spiral model, however, recognises four separate levels in technical software modification. The AMES model covers also the strategic decisions and justifications for organising and planning change activities.

6.7 Summary

This chapter proposes a model for software change management processes. The model is derived from two sources: change models presented in the literature and three case studies, in which the change processes of three organisations were studied. The model is compared with the change models presented in chapter 3.5.

The model proposed here is intended to be used together with the problem classification framework presented in the previous chapter (Chapter 5) for defining a new change management solution for the organisation. First, the problems of the organisation in question should be analysed using the problem classification framework, and then the generic process model should be instantiated to respond to the specific problems identified. The next chapter (Chapter 7) demonstrates how this was done in one case implementation.

7 Implementation

7.1 Overview

This chapter uses the proposed problem classification and process model in analysing the change management problems and requirements, and defining new change management processes in one case study. The organisation studied in the case study presented in this chapter is not one of the organisations described in the appendices. The analysis of the initial status of change management practices and needs is described, and an implementation solution is presented. The new practices had been in use in the organisation for approximately three years at the time of writing this thesis. A summary of the evolution of the change management environment during the time of its operation from the first release to writing of this thesis is given at the end of the chapter.

7.2 Operational organisation

The case organisation develops electronic multi-processor products, in which software is a very crucial product element. The software development projects are geographically distributed. The sizes of the software development teams vary between different geographical sites, ranging from tens to hundreds of people.

7.3 Process management

The change management processes in the organisation were sub-processes within the software development and maintenance processes. There were two main methods for managing the change management processes in the projects:

- Reviews of the change requests, and
- Monitoring the change requests.

7.3.1 Reviews of change requests

The practices for change request reviewing varied considerably between projects and project types. Three ways of reviewing the change requests were identified:

- *Subproject manager meetings*. The project managers of the different technology subprojects of the same product development project reviewed the changes currently processed by the software project once or twice a month. The purpose of the review was to distribute change information between the subprojects, discuss open issues, and make decisions about changes that have major impacts on the product developed in the subprojects.

- *Test group reviews*. The test groups reviewed the error lists for sharing information, planning test cases and testing sessions and analysing the effectiveness and success of the test sessions.

- *Project meetings*. Most projects reviewed the change requests in project meetings, which took place weekly or monthly. The purpose of the reviews was to support project planning and monitoring and to discuss the change requests.

7.3.2 Monitoring the change requests

The change requests were monitored by:

- *Software designer*, who was responsible for the software component. The responsible person received error information by E-mail, and defect information in reviews, where the person who was responsible of the software component reviewed was usually present. Both errors and defects were recorded using the error report tool, and the software designers could monitor defects and errors using the tool.

- *Project and subproject managers*, who monitored change requests in order to follow the status of the project in making project estimates and planning new tasks and milestones.

- *Chairman of the review session*, who used the error report tool to monitor whether or not the author of the document had corrected the defects found in the review session.

- *Test group*, who monitored the errors recorded in the error report tool. The test group used the error information in planning test cases and testing sessions, identifying valid regression testing cases and analysing test sessions.

7.4 Quality responsibilities related to change requests

The quality responsibilities for the changes depended on the software development phase or event when the request for change was generated, and the type and impacts of the change. Quality responsibilities can roughly be described for the following classes:

- Change request created in a technical review.

- Change request detected in testing.

- Other change requests, such as requests received from marketing, production, standard development groups, internal improvement proposals, etc.

7.4.1 Technical review

There were two kinds of quality responsibilities related to the change requests detected in the technical reviews: the author of the document was responsible for implementing and testing the modification and the chairman of the review session was responsible for re-inspecting the module or the document.

7.4.2 Testing

There were four main test phases: unit, integration, laboratory and acceptance testing. Errors found in the unit testing phase were not recorded. The quality responsibilities related to the errors found in the testing phases were rather complicated, since the severity, scope and type of the change affected them. Generally speaking, the person who found the error was responsible for record-

ing the error and distributing it to all interested parties. The person responsible for the document in question, i.e. the software component under testing, usually performed the analysis, implementation and testing of the modification. However, sometimes the scope of the error was not that obvious, or the modification required changes in several parts of the product, and the analysis and implementation responsibilities had to be discussed and shared. If the person responsible for the document evaluated the error to be critical, he passed the analysis information to the software project manager for implementation decision and planning. The change was usually validated in the next testing session, where it was not supposed to occur again, and it should not have generated new errors related to the correction. The testers were responsible for testing the corrections. Some projects provided the error lists from the error report tool to the test group for regression test planning. The software project manager was responsible for monitoring that the errors found were corrected and for organising the resources and schedules of the correction activities.

7.4.3 Other change types

Since the practices and responsibilities for managing other types of change requests had not been defined, the product project and software project managers were responsible for monitoring, initiating and planning them.

7.5 Sources of change requests

The sources of change requests are illustrated in Figure 19.

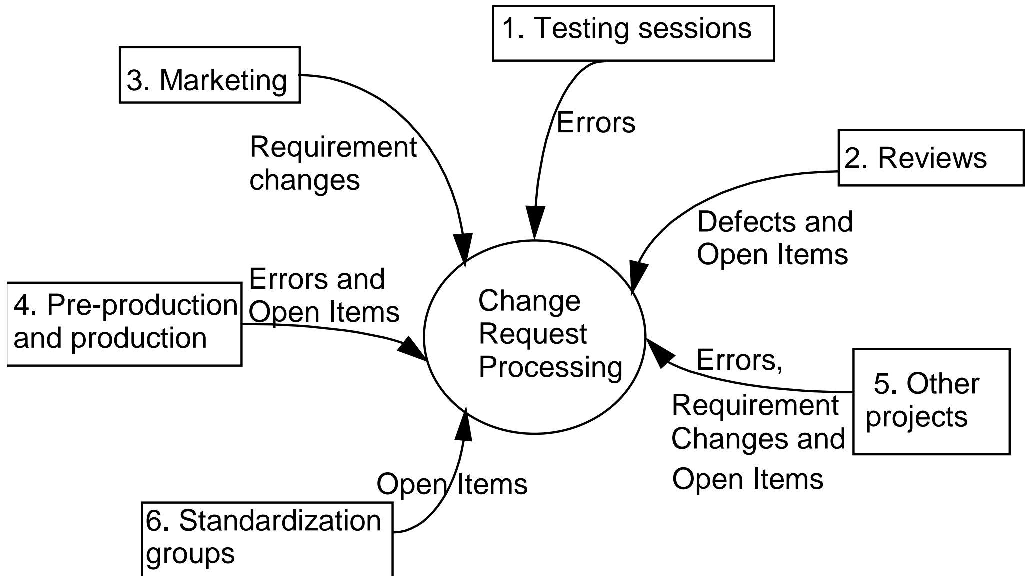


Figure 19. Sources of change requests.

The following list includes a description of the media or format of the change requests received from the different sources:

1. Testing sessions

- Module testing. Changes or change requests were not recorded. The author of the module tested his own work and corrected the errors found immediately.

- Integration and laboratory testing. Change requests were recorded using the error list of the error report tool. The change request was updated and monitored using the error report tool.

- Acceptance testing. Practices varied. Some errors found were recorded using the error report tool as the errors were found in the integration testing phase. Some of them were received by the person who was responsible for the module or by the subproject manager by E-mail, paper or phone. The person responsible for the module should have stored the change informa-

tion received in informal format using the error report tool, but this did not always happen.

2. Reviews

- Source code reviews. Change requests were recorded using the defect list of the error report tool. The requests were updated during the modification process and monitored using the error report tool.

- Specification and design reviews. Practices varied. Some projects used the defect list of the error report tool, some of the projects added the defects that were found to the review minutes (a textual document written using a word processor).

- Management and planning reviews. The defects detected in management documents were recorded in the review minutes.

3. Marketing and other instances triggering change at the requirement level. These were handled informally. No formal channels for receiving change requests existed. Change requests were usually received in informal format by E-mail, in meeting discussions, or coincidental discussions. One of the projects had drawn up a change request form but it had not received any requests in that format.

4. Pre-production and production. The person who was responsible for the pre-production testing wrote a 'Pre-production Analysis' report, in which the results of the tests were reported. The subproject manager received this report by E-mail.

5. Other projects. The change requests between projects and subprojects were usually transmitted and negotiated in meetings, or by discussions between project or subproject managers. The error lists of the projects were accessible by all projects, but the distribution of error information between projects or subprojects was not supported by the tool. Error lists could be used as a reference in meetings and discussions.

6. Standardisation groups. The new standards have to be followed continuously by the product experts in order to identify standardisation issues that affect the product. If such an issue is found, the product expert creates an open item about the issue.

7.6 Description of the change management process

Since process support existed only for errors found in testing and defects found in reviews, they were the only change types which had defined processes. The processes were documented and used by the error report tool. The processes are illustrated here by using state transition diagrams. The two graphical elements of the diagrams are:

1. Rectangles, which represent the state.

2. Arrows to indicate transitions between states.

The input event (above the line) and a set of output events (below the line) appear next to the arrows. Figure 20 illustrates the error management process. The state transition diagram presented in the figure was part of the error report tool documentation [Case material].

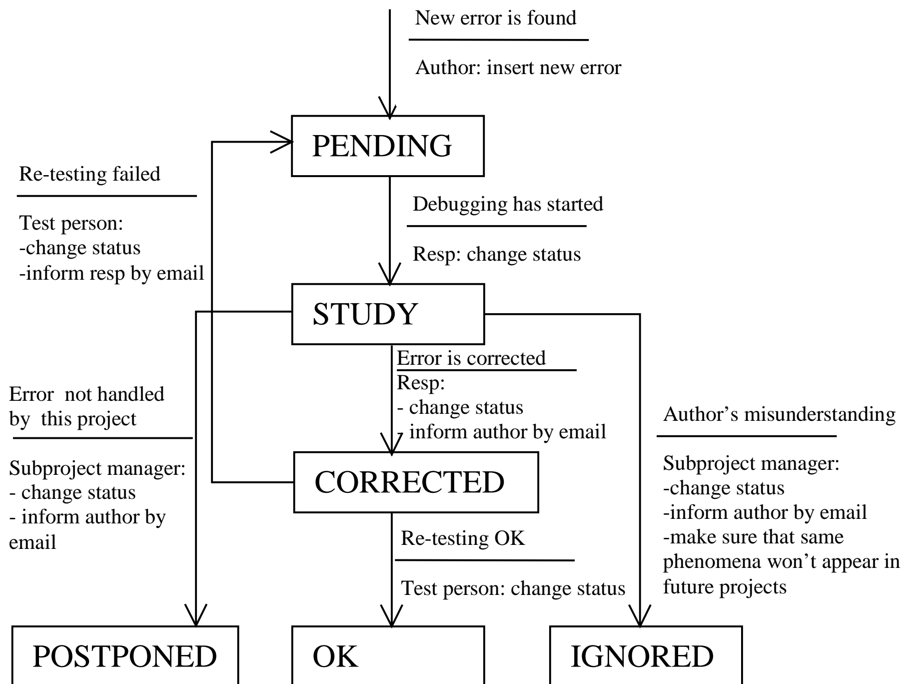


Figure 20. Error life cycle used by the error reporting tool. (Figure taken directly from tool documentation [Case material]).

The defect life cycle used by the error reporting tool is illustrated in Figure 21. The same life cycle was followed even when the error reporting tool was not used for recording defects.

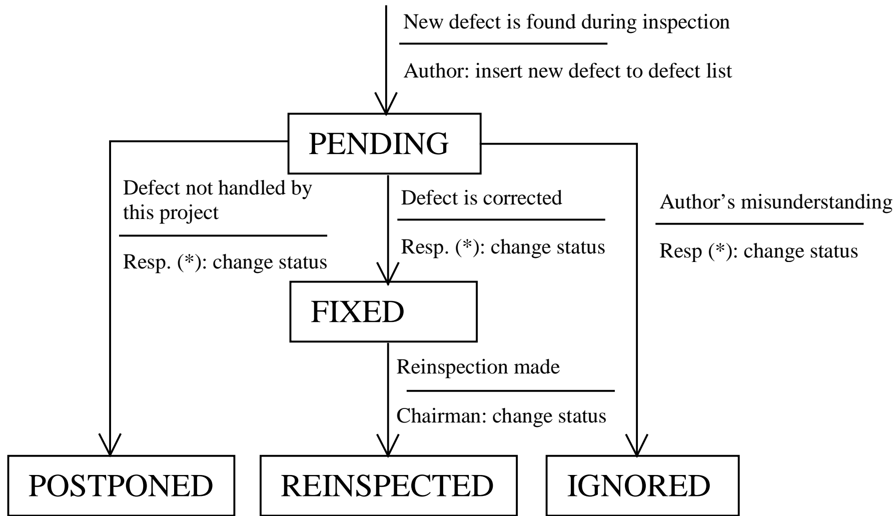


Figure 21. Defect life cycle used by the error reporting tool. (Figure taken directly from tool documentation [Case material]).

The processes described in Figure 20 and Figure 21 are the ones which were used by the change tools in use at the time of the interviews. However, the interviewees thought that these processes did not fully describe all types of changes and all activities of the change processes. The following main phases (illustrated in Figure 22) were identified by the interviewees:

1) *Detecting and recording the modification need.* This phase included such activities as location of the change in the product, creation of change request, distribution of the change request by e-mail, phone, meetings, etc.

2) *Decision about the implementation*. This phase included activities related to analysing the impacts of the change in the product, project and other projects, generation of solution proposals, risk analysis, organisation of the modification tasks and change request review meetings.

3) *Implementation* of the change, i.e. changing the documentation and code and testing the change.

4) *Verification* of the change, including a review or other acceptance procedure for the change request and regression testing.

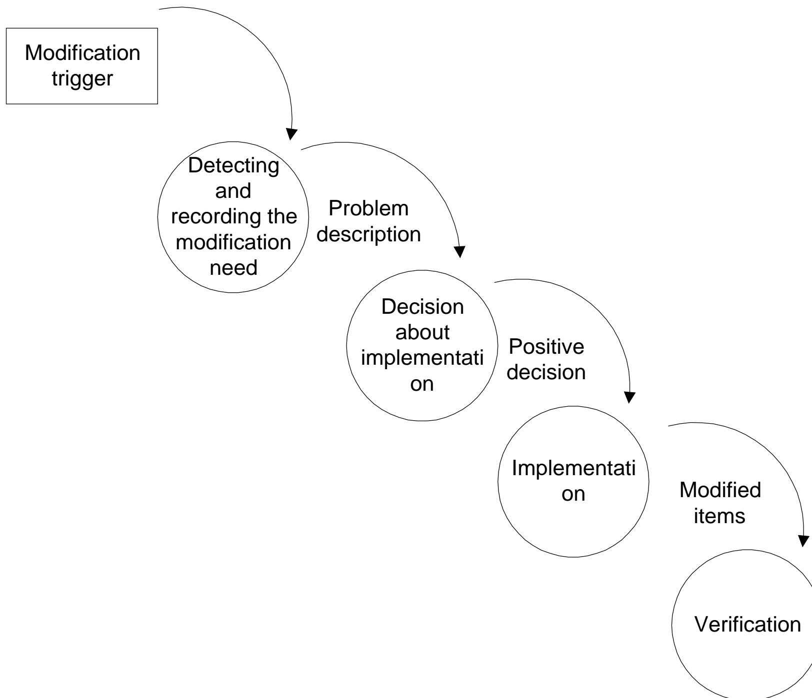


Figure 22. Main phases of the change management process identified by the interviewees.

7.7 Existing change management support

The main tool used for change management was an error recording tool used for recording testing errors, review findings and improvement proposals. Some projects had drawn up forms without tool support for recording requirement-level changes. In addition, other technology groups had their own tools for managing change requests. Examples of these tools were tools used for managing user requests and problems by user support departments, production fault databases, etc.

The best supported change process was the error management process. Errors were recorded quite rigorously in all projects using common procedures. Errors were defined as:

Deviations from requirements or required behaviour and the actual behaviour of the system found in the testing phases or identified after the delivery.[1]

The deviations found in the review sessions were called defects. Defect management was also supported, but the practices varied between the projects. Some projects used the error-recording procedures to manage defects found in reviews, some used project-specific document templates for creating a combined meeting minutes and defect report, and used the same document for defect follow-up.

In summary, the company called deviations found in testing phases 'errors', and deviations identified in the review sessions 'defects'.

Both errors and defects (as defined in the organisation studied) fall into the definition of 'defect' according to the generic change management model. Our generic model does not separate defects according to their identification phase, as was done in this organisation.

Miscellaneous change requests, such as improvement proposals, wishes for new features, etc. were called open items. Support existed for handling these changes, but the practices varied a lot between projects. The definition of 'open item' differed between the projects. Some managed requirement change requests using open item procedures, other projects included internal improvement proposals or unclear error situations in the 'open item' category.

The most informal change process was the management of changes received from outside the software project during the development time, e.g. from marketing and other software or hardware projects.

---

[1] Quote from company error-recording guidelines. Internal document, exact reference cannot be made due to confidentiality reasons.

7.8 Problems and improvement proposals related to change management

This section presents the problems, improvement ideas and requirements put forward by the interviewees with respect to software change management in the organisation. The problem classification presented in chapter 5.2 is used for classifying the identified problems.

7.8.1 Effectiveness problems

Problem 1. Ineffectiveness of the technical reviews

Technical document reviews faced two problems: (1) it was difficult to get the experts needed for achieving the best review to a review, since they were busy and had a great deal of work to do, and (2) the participants of the review session seldom had spent enough time for reading the reviewed documentation.

During the interviews the improvement idea was put forward that the review tool could support entering defect proposals prior to the review meeting, allowing persons not able to attend the review session to enter their findings, if the review session would focus on handling the pre-recorded defect proposals. This would motivate reviewers to be better prepared for the meeting. Defect proposals would be preliminary findings of the reviewers, which are further discussed and refined into defects in the review meeting, if they are found to be real deficiencies.

7.8.2 Communication problems

Problem 2. Lack of training courses and introduction material on change management practices

There had not been enough training or self-learning material for the change management methods and tools. This resulted in inconsistencies in the change management practices between projects.

Problem 3. Informal sources of change requests

No formal change request document or form existed for change requests received from outside the software project. These change requests were received in free-text form, e.g. E-mail, personal discussions, during meetings, etc.

Problem 4. Lack of sharing of change requests between projects and subprojects

Each project and subproject had their own lists of change documents, and no support for sharing and distributing the change documents existed. This caused problems especially when the modules were shared between projects. The contents of the change documents were very similar across projects. The change document included the original change request completed with additional information accumulated during the change process, i.e. information about the solution, decisions leading to the end result and acceptance details.

7.8.3 Analysis and location problems

Problem 5. Impracticality of using the severity classification for UI (user interface) projects

The error severity had been analysed using the scale of four presented in Table 15.

Table 15. Error severity classification.

****	Major, an error that does not conform to requirements and/or will cause a product failure
***	Causes malfunctions visible to user
**	Causes malfunctions invisible to user
*	Minor, an error that is not likely to cause a product failure

The classification had been found to be unpractical in the user interface (UI) subprojects, since the classification was based on the visibility of the error to the user and basically all UI errors were visible to the user. The deviations between errors recorded in each severity class between different types of projects is presented in Table 16.

Table 16. Distribution of errors between severity classes and different project types.

Severity class	UI projects	Other SW projects	Other projects
****	40 %	30 %	10 %
***	50 %	30 %	10 %
**	10 %	30 %	40 %
*	0 %	10 %	40 %

Problem 6. Modification requests in the wrong lists

Some software designers found it annoying when testers recorded errors which were not actually errors, but rather improvement ideas. In one project 34 % of the errors recorded in the error list had been ignored because they were not real errors, i.e. inconsistencies between the requirements and the actual behaviour. Software designers felt that these recordings made the error statistics look worse and generated extra work for subproject managers, who had to check all the recordings and ignore unnecessary error recordings.

Problem 7. Postponed change requests were lost

When the project postponed the change request, i.e. it did not implement the change, but forwarded it to the later projects, the postponed change requests were often lost. The change request lists were owned by the project, and the new projects started with an empty change request database.

Problem 8. No feedback loop

The change data was not used to feed process-related information to the ongoing or new projects. The change histories of the projects were not systematically analysed in order to identify root causes for typical defects, providing information of the typical amount of rework and changes for project estimation, etc.

7.8.4 Traceability problems

Problem 9. Poor traceability between changes and software items

The traceability between the changes and affected software items was poor. The projects usually recorded change histories into document headers. The contents of the document headers were similar in different types of documents (e.g. design documents, source code files, test reports), only the format differed due to implementation issues (e.g. design tools generated document headers in a different format than the headers written manually with text editors or word processors). The change history had a description why the new version was created, but not necessarily a link to the change document which initiated the modification. The location of the correction had to be entered into the change documents, but the field was a free-format text field, and therefore it could include anything from file listings to subsystem identifier. The loss of traceability information complicated, for example, release building and component sharing between projects. In release building, the traceability information would have been needed to ensure that certain modifications had been implemented in the components included in the release. When a component was shared by several projects, the traceability information would have been necessary in distributing error corrections or other changes between projects.

Problem 10. No link between the change document and the modified items

The link indicating which software items were impacted or created because of the change was not stored. The link would have been valuable, for example, in assuring that the software release included certain error corrections or other modifications. The link would also provide valuable measurement data for quality management purposes, for example, it could provide information about the error-prone modules.

Problem 11. Inconsistencies between abstraction levels

No support was provided for defining how the change in one abstraction level affected the lower or higher abstraction levels of the system. For example, when changes in the source code were implemented, there was no support for defining which design parts or test data parts should have been modified to keep the sys-

tem consistent. This resulted in inconsistencies within the software system, since the changes were not propagated to other abstraction levels when one level was changed.

7.8.5 Decision-making problems

No decision-making problems were found in this case organisation.

7.8.6 Tool-related problems

Problem 12. No support for the generation of a change history list for the product review authorities

The product review authorities required a list of changes which had been implemented after the previous product review or acceptance. The tools used for change management did not support generating this list, although it would have been a rather straightforward, easily automated task. At the time of the case study, the list had to be created manually. The manual creation process was very time-consuming and error-prone.

Problem 13. Wrong input values

The error report tool did not check the validity of the user-entered values and it did not give any guidance to the user regarding what the entries should contain. For example, the error list included a field “Location”. The purpose of the field was to identify the location of the error in the software (i.e. the field should contain a list of files, documents and modules); however, in the acceptance testing phase the field name can be easily mixed with the geographical location, i.e. the field sometimes had input values expressing the geographical location where the error had been found, such as “Customer premises in Helsinki”. Furthermore, as the field was a free-format text field, the file listings were prone to spelling mistakes and inconsistencies.

As the tool did not give any guidance to the user and all the input fields were free-format text fields, the entered values were often inconsistent with each other, resulting in problems when making statistical analyses or summary re-

ports. For example, the fields which required entering time information had input values ranging from minutes to the phrase ‘not much’.

Problem 14. Input fields which are not used anywhere

The error report tool had several input fields which were not used anywhere. For example, the time used for fixing the error had to be entered by the user, but it was not used in summary reports or monitoring activities. This resulted in unreliable input values, since the users of the tool knew their input values were not used anywhere and were not motivated to enter them.

7.9 Description of the implementation solution

The results of the change management analysis were used as an input to define new change management processes and process support for the case organisation.

7.9.1 Vocabulary

The vocabulary used in the process descriptions instantiated in the case study is the following:

- **Process.** Process is a set of phases followed in order to manage a change in a software system. An input to the process is a trigger for a change, and outputs are either a closed change documents or modified software items, if the change request is accepted for implementation.

- **Phase**. The phases form the generic parts of the change management processes. The phases are the same for all change process types. One phase is typically performed by one person, and it has defined inputs and outputs. The phases also define the basic change document life cycle, i.e. the status of the change document is updated during each phase. As an exception, when the life cycle of the change document is relatively short, some statuses may be updated simultaneously, when the phases are performed together with each other.

- **Step**. Phases are divided into steps. Steps are specific for each change process type. Step descriptions define the actual activities performed in that particular phase. The steps are documented using **Step-Action** tables, where the "**Step**" column gives a name for the step, and the "**Action**" column describes the action implemented by each step.

- **Session**. A session is an event, which takes place in a certain period of time. A number of phases may be performed in a session. Examples of sessions are testing sessions, review sessions, project meeting sessions, etc.

- **Review**. A review is a meeting, where the participants collect, discuss and analyse the defects they have identified in the documents or document parts distributed before the meeting.

7.9.2 Instantiation of the generic processes in the case study

The generic types of change processes presented in chapter 6.3.2 were used in defining the change processes in the organisation. The following processes were defined:

- Error management process (corresponding process in the generic model: Defect correction process)

- Defect management process (corresponding process in the generic model: Trivial defect correction process)

- Open item management process (corresponding process in the generic model: Improvement proposal)

- Requirement-level modifications (corresponding process in the generic model: Requirement-level modification)

Roles

The change management roles are described in Table 17. The role names will be used later on in the process description sections.

Table 17. Change management roles.

Role	Responsibilities of the role
Initiator	Identifies the change and writes the change request (phase 1 of generic change management process)
Analyser	Analyses and studies the change request and writes the analysis notes (phase 2 of generic change management process)
Evaluator	Decides whether the change will be implemented, ignored or postponed (phase 3 of generic change management process)
Author	Implements the change and links the affected documents to the change notice and writes implementation notes (phase 4 of generic change management process)
Approver	Approves the change and writes the validation notes (phase 5 of generic change management process)

Error management process

The error management process follows the generic change management process. The phases are further divided into smaller steps, which are unique to error management. The first phase, identification, includes steps described in Table 18. The identification phase is carried out by the initiator of the change.

Table 18. Steps of the identification phase of the error management process.

<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>1.
Identification</td><td>Identify the behavior or feature, which may be an error.</td></tr>
<tr><td>2.
Location</td><td>Locate the cause of the behavior as accurately as possible. Use the following check-list:

• Check the version numbers of hardware and software components.
• Is the error located in hardware or software?
• Which component or module contains the cause of the error?</td></tr>
<tr><td>3.
Analysis</td><td>Analyse the error. Check the following things:

• Did the error occur because of a misunderstanding or misuse?
• Are there external factors which may have interfered with the results?
• Are you able to repeat the error?
• Is this an error or an improvement proposal? If an improvement proposal, create new open item</td></tr>
<tr><td>4.
Evaluate the severity</td><td>Evaluate the severity of the error. Use the following four classes:

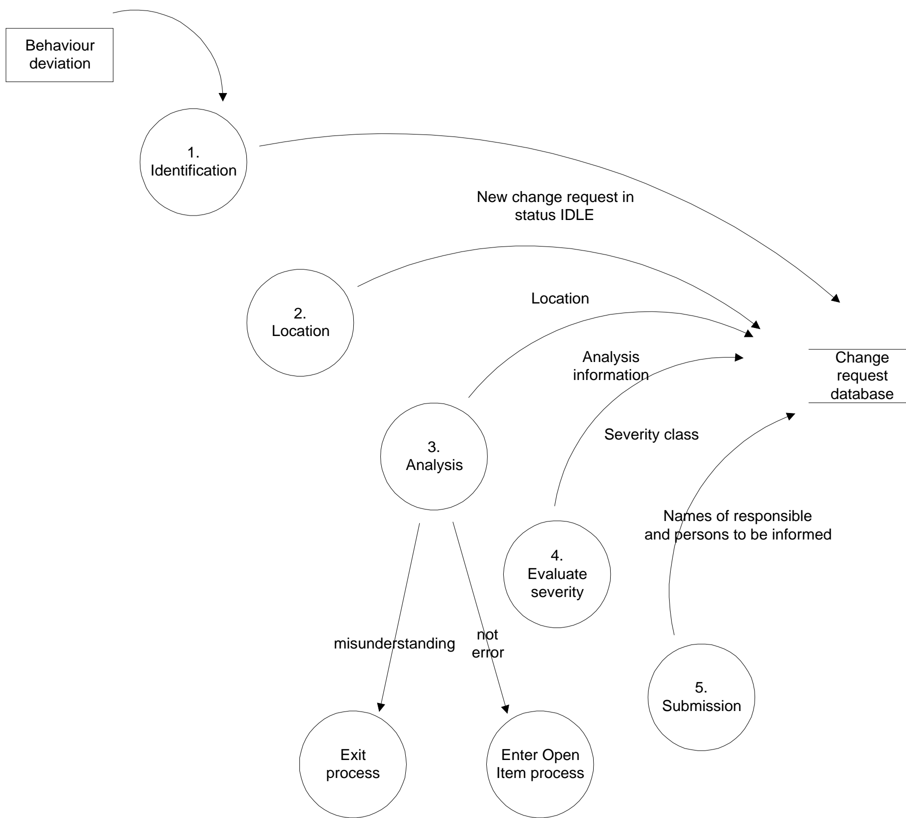
Severity class — Description
**** — Major, an error that will cause a product failure.
*** — Causes malfunctions that the user is likely to notice.
** — Causes malfunctions that the user notices only in some exceptional cases.
* — Minor, an error that is not likely to cause a product failure

The severity class descriptions can be tailored for your project type, so check the class descriptions from your system instantiation.</td></tr>
</table>

Table 18. Continues.

Step	Action
5. Submission	Submit the change request to the people in the following list: 1. the person responsible for the module, 2. the subproject manager, or 3. the project manager. In addition, define the persons you want to inform about the error request.

The steps of the identification phase are illustrated in Figure 23.



.

Figure 23. Steps of the identification phase of the error management process.

In the identification phase the change requests are forwarded to the person responsible for analysing the error. If the initiator does not know the person responsible, he will send it to the subproject manager. If the initiator is not sure which subproject is responsible for the part causing the error, for example when it is not clear if the error is caused by a software or hardware fault, he forwards the created change request to the project manager. The subproject or project manager then evaluates the change request and forwards it to the right people. If the person responsible for the analysis is wrongly defined, the subproject or project manager will appoint a new person. The responsible person will receive the change request by E-mail, and the change management tool will provide lists of all change requests assigned to the individuals. The person responsible begins

analysing the error. The steps of the analysis phase are described in Table 19 and illustrated in Figure 24. The analysis phase is carried out by the analyser (see roles in Table 17).

Table 19. Steps of the analysis phase of the error management process.

Step	Action
1. Evaluate responsibility	Check if the modification request has to be submitted to someone else • repeat until the right person has been found
2. Inform other projects	Evaluate if the other projects should be informed about the error. Send E-mail to those to be informed.
3. Update status	Set the status of the change request to STUDY.
4. Re-evaluate the severity of error	Check if the severity classification defined by the author of the change request is appropriate.
5. Analyse the causes	Analyse the causes of the error.
6. Generate solution proposals	Generate several possible proposals for eliminating the error.
7. Impact analysis	Analyse the impacts of the solution proposals. Analyse the impacts on: • documentation • hardware components • software components • test data The documents that need changing are called affected documents.
8. Estimate resources needed	Make preliminary estimation about the working time needed for implementing the change.

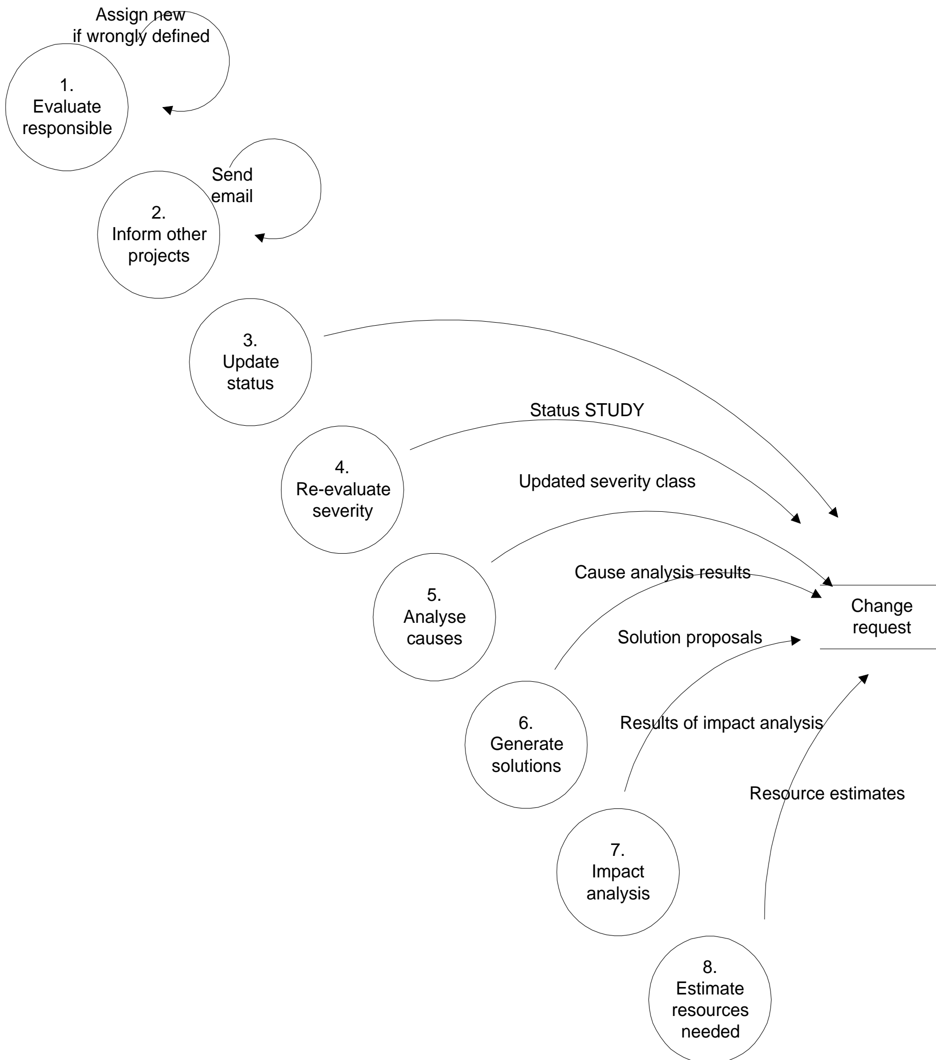


Figure 24. Steps of the analysis phase of the error management process.

The goal of the analysis phase is to provide all necessary information for making the implementation decision. The steps of the implementation decision phase of the error management process are described in Table 20 and illustrated in Figure 25. The person making the implementation decision is called the evaluator (see Table 17).

Table 20. Steps of the implementation decision phase of the error management process.

<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>1.
Decision</td><td>Decide, whether to

a. implement the change
b. ignore the change
c. postpone the change for consideration in later projects

If the erroneous software module is used by several projects, define
a. who will correct the error
b. how the corrected version will be distributed to the projects
c. how the correction will be validated by the different projects</td></tr>
<tr><td>2.
Update status</td><td>Update the status of the change request as follows:

<table><tr><th>Decision</th><th>Set status to...</th><th>Then...</th></tr><tr><td>a.</td><td>ACCEPTED FOR IMPLEMENTATION</td><td>go to step 3</td></tr><tr><td>b.</td><td>IGNORED</td><td>Explain why the error was ignored.
Exit process.</td></tr><tr><td>c.</td><td>POSTPONED</td><td>Explain why the error was postponed.
Define who should examine the error.
Exit process.</td></tr></table></td></tr>
<tr><td>3.
Set schedule</td><td>Set the deadline and schedule for implementation.</td></tr>
<tr><td>4.
Set resources</td><td>Set the resources for implementation.</td></tr>
</table>

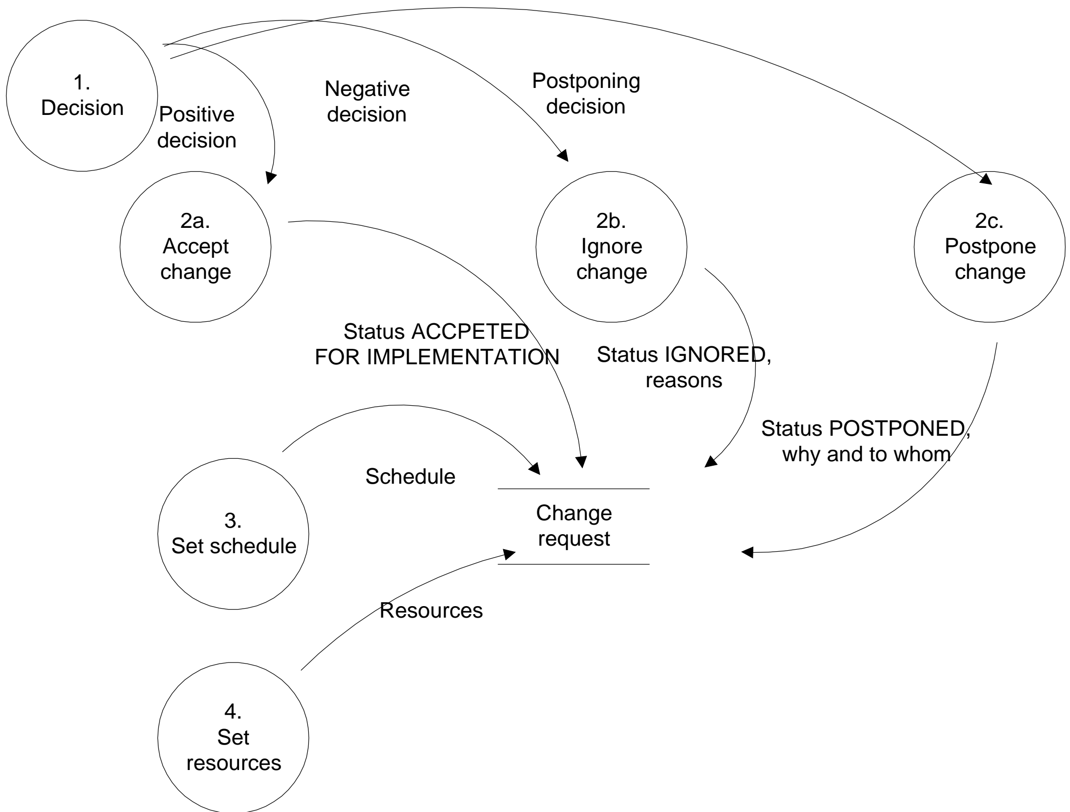


Figure 25. Steps of the implementation decision phase of the error management process.

Depending on the implementation decision the change process will either continue its normal execution path, or be terminated, if the decision is not to implement the change in this project. In the case of a positive implementation decision, the change proceeds to the implementation phase. The resourcing and scheduling of the implementation are defined in the implementation decision phase. The steps of the implementation phase are described in Table 21 and illustrated in Figure 26. The implementation phase is carried out by the author (see Table 17).

Table 21. Steps of the implementation phase of the error management process.

Step	Action
1. Modify software	Correct all affected parts of the software: documentation, test data, source code, scripts etc. When you return the modified items to the version control system, you must link the modified items using the Change Notice. The principle of the Change Notice is presented in Figure 31.
2. Describe imple-mentation solu-tion	Write a description of the implementation solution used for correcting the error.
3. Record working time	Record the working time used for the modification.
4. Update status	Set the status of the change request to IMPLEMENTED.

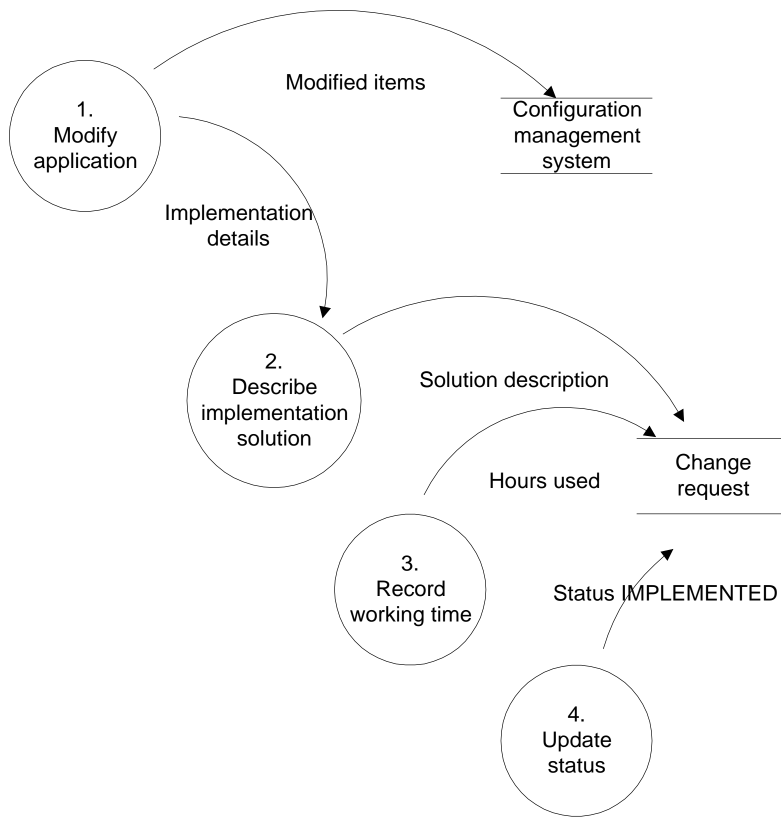


Figure 26. Steps of the implementation phase of the error management process.

The last step of the error management process includes the activities aiming at validating the correctness of the implementation and informing about the correction. The steps of the validation phase of the error management process are described in Table 22 and illustrated in Figure 27. The validation phase is carried out by the 'approver'.

Table 22. Steps of the validation phase of the error management process.

<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>1.
Validate modification</td><td>Check the correctness of the implementation.

a) If the corrections are OK, go to step 2.
b) If problems are found, go back to the implementation phase, and set the status of the change request to ACCEPTED FOR IMPLEMENTATION (an input state for the implementation phase).</td></tr>
<tr><td>2.
Check impacts</td><td>Check that the modification made did not have any negative impacts on other parts of the application.

In the case of code changes, perform regression testing. In the case of document changes, check the consistency within the document and between other documents.

<table>
<tr><th>If tests reveal new defects...</th><th>Then...</th></tr>
<tr><td>…which are impacts of the modification and should have been noticed in impact analysis</td><td>…go back to the implementation phase. Set the status of the change request to ACCEPTED FOR IMPLEMENTATION</td></tr>
<tr><td>…which are not impacts of the modification</td><td>…record a new error</td></tr>
</table>
</td></tr>
<tr><td>3.
Update the status</td><td>Set the status of the change request to OK, after the testing and regression testing phases have been successfully performed.</td></tr>
<tr><td>4.
Inform other projects</td><td>Determine if there are other projects that should be informed about the correction. Inform them by E-mail.</td></tr>
</table>

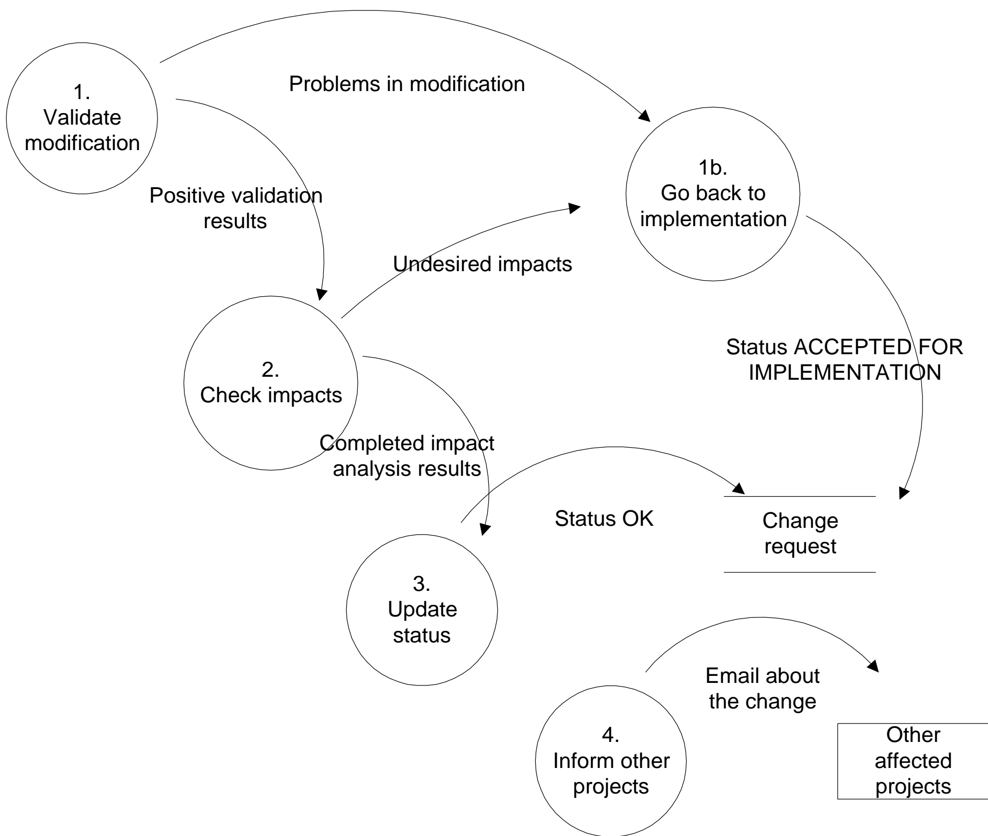


Figure 27. Steps of the validation phase of the error management process.

Defect management process

Defects are received from two primary sources:

- technical reviews, where the document reviewed is a software work product (specification, design, source code, test plans and results, etc.) and

- milestone reviews, where the document reviewed is a project management document (project plan, final report etc.)

The defect management process follows the process for trivial defect correction (Figure 15). The first three phases of the generic change management process (Figure 14) - identification, analysis and implementation decision - are performed in a review session. The change request is created after the review session and therefore the life cycle of the change request is shorter than in the case of errors and open items. Usually all the defects recorded will be implemented, because the implementation decision is done in a review. If the project wants to, it can use the status "ignored" of the generic change management process to ignore defects recorded in a review.

Defects are identified in review sessions. The participants of the review analyse the change proposals, and decide if the change proposal is a defect and whether it should be implemented or not. The change proposals can be presented by all participants and they are based on the document study made before the review session. The secretary of the review writes the minutes and records the defects and open items raised in the review.

The initiator of the defect (see roles in Table 17) is the secretary of the review. He records all defects identified in the review. The reviewers as a group act as change analyser and evaluator. The steps taken in the review session itself are presented in Table 23.

Table 23. Steps of the review session.

Step	Action
1. Collect the change proposals	The participants present the change proposals they want to propose. The proposals are based on the reviewers' reading of the reviewed documents before the review session.
2. Analyse the change proposals	Analyse whether the change proposals are deviations from requirements, improvement proposals, misunderstandings etc.
3. Classify the change proposals	Classify the change proposals into two classes: **a.** Open items, if the proposal is not a defect but should generate a change proposal anyhow **b.** Defects
4. Implementation decision	Decide whether to implement the changes or not. A defect record is created ONLY for defects which are decided to be implemented. For open items, the implementation decision is not done in the review meeting, so a change document is created for all proposed open items.
5. Set resources	Set implementation resources and possibly deadlines.

The steps performed after the review by the secretary are presented in Table 24.

Table 24. Steps to be taken after the review session.

Step	Action
6. Write minutes	Write the minutes of the review. The status of the minutes is OPEN.
7. Record defects and open items	Record the defects identified in the review session. The defects are recorded using the form in the Change management tool.

The steps related to the review session are illustrated in Figure 28. These steps cover the first three phases of the generic change management process, i.e. the identification, analysis and implementation decision steps, since they all are taken in the review session.

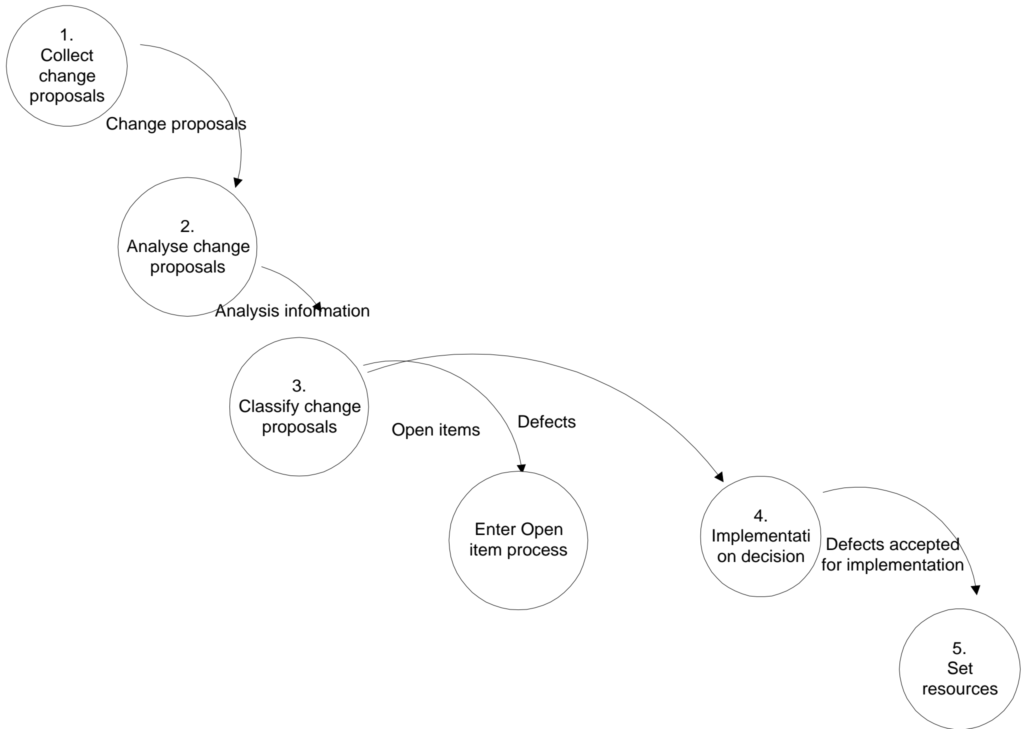


Figure 28. Identification, analysis and implementation decision steps of the defect management process.

The implementation is usually done by the person who is responsible for the document. If that is not the case, the implementation resources must be determined by the reviewers in the review session. The implementation steps are described in Table 25 and illustrated in Figure 29.

Table 25. Implementation steps of the defect management process.

Step	Action
1. Identify affected documents	Identify which documents are affected by the modification.
2. Implement	Modify all affected documents. When you enter the modified items into the version control tool, they have to be linked with the change notice in the version control tool. (see Figure 31)
3. Update status	Set the status of the change request to IMPLEMENTED.

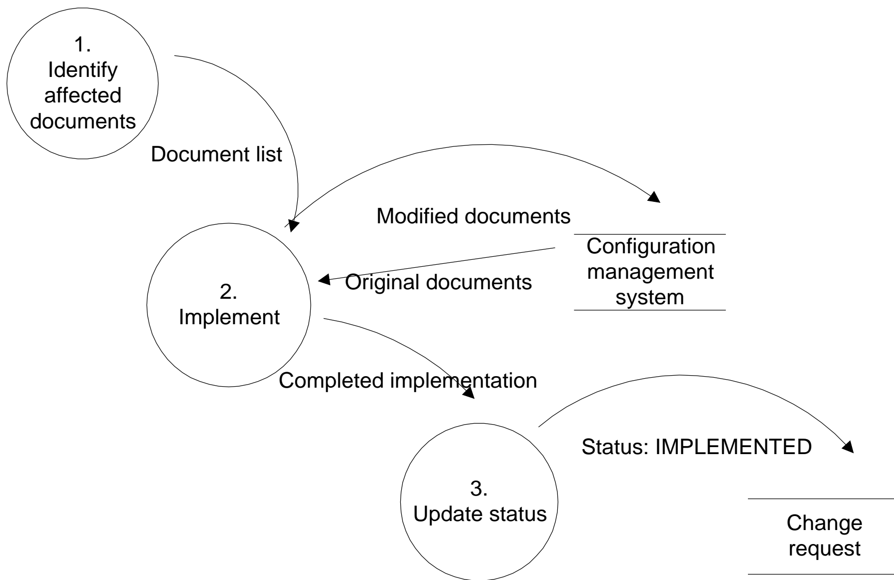


Figure 29. Implementation steps of the defect management process.

The approver of the defect implementations is usually the chairman of the review session, if not otherwise decided in the review. The steps of the validation phase are described in Table 26.

Table 26. Steps of the validation phase of the defect management process.

Step	Action
1. Check	Check that the defect has been corrected and the correction is valid.
2. Update status	Set the status of change request to OK.

After all defects initiated by the review meeting have been closed, the status of the meeting minutes is changed to CLOSED. This can be done either automatically by the change management system, or manually by the chairman of the review session.

Open item management process

The improvement proposals, ideas, open issues, etc. are managed using the generic change management process. The problem analysis indicated that the process for managing open items itself did not play a very crucial role for these types of change requests. The most important issue was regarded to be that there was a place where to collect and record open ideas and improvement proposals. Typical sources for open items are project members, testing groups, standardisation groups and end users. Open items may also be created by changing defects or errors which are not deviations from requirements but improvement proposals to open items.

Requirement-level modifications

Defining processes for requirement-level modifications was not included in the project in which the author of this thesis was involved. The requirement-level change management process was studied and defined using the requirements engineering process.

Two layers of requirement changes had to be supported: a process for managing project-level requirement changes and a process for managing product-level requirement changes. The project-level requirement changes are changes, which

arise internally in the software development cycle; for example the inability to meet the required memory size constraints or performance requirements. The product-level requirement changes are changes to the software requirements which are raised by an external requirement change request.

Both processes follow the main phases of the generic change management process, with some additions. The main difference is the possibility to create separate implementation proposals from the change request in the analysis phase (phase two of the generic change management process) as illustrated in Figure 30. The implementation proposals describe the possible implementation solutions responding to the change requirements indicated by the change request.

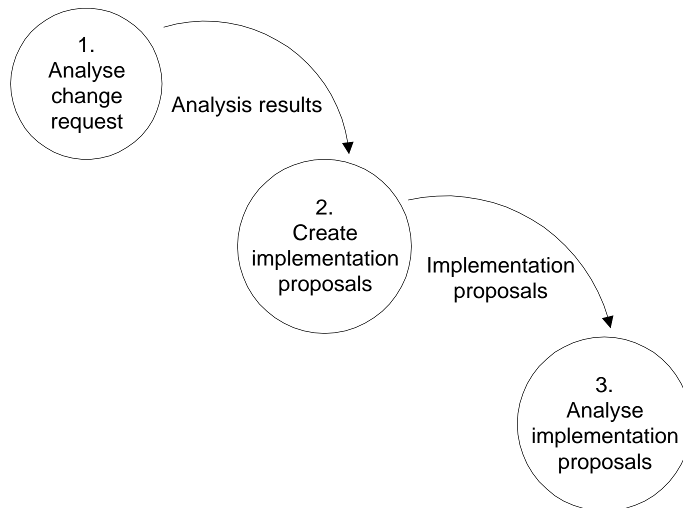


Figure 30. Analysis steps of the requirement level change process.

The steps of the analysis phase are described in Table 27.

Table 27. Steps of the analysis phase of the requirement level change process.

Step	Action
1. Analyse Change Request	Analyse the change request from the following viewpoints: 1. Check that the right person has the responsibility. 2. Check if you agree with the severity classification defined. 3. Analyse what the possible benefits, problems and risks associated to this change request are.
2. Create Improvement Proposals	Create one implementation proposal for each influenced technology sector. Forward the implementation proposals to the persons responsible for the technology sectors.
3. Analyse Improvement Proposals	The responsible persons should analyse the following aspects of the improvement proposals: • How should this change request be implemented? • What are the influences of the change request to project schedules? • What are the influences to interfaces? • What are the affected documents? • What is the estimated work amount related to the change? • What was the actual work amount used for analysing the improvement proposal? Make a recommendation whether this particular implementation proposal should be accepted, accepted with modifications or rejected.

The project-level requirement modifications also follow the generic change management process, with special features on decision-making and communication between technology groups. The decision-making process is described in Table 28.

Table 28. Steps of the implementation decision phase of the requirement level change process, when the need for the requirement change comes from within the project.

Step	Action
1. Initial internal decision	Decide if the requirement change is needed.
2. Inform customer	Inform the customer how the change will affect the customer interface, if the change has visible effects on customer interfaces.
3. Review customer interface	Review the changes to the customer interface with the customer. As a result of the review, forward the requirement change to implementation.

7.9.3 Selection of tool environment

The initial goal of the project was to examine the possibilities of using the new configuration management tool for supporting the change management process. However, when the process analysis phase progressed, other implementation solution options arose. The tool selection was carried out by the same group of people who defined the change process instantiations.

The tool selection group evaluated the major benefits of using the same tool for software configuration management and change management to provide the easiest way of generating and managing the traceability links between the change documents and the software items. The changes could be linked with the software items when doing normal version control operations, such as fetching a file version for modification. The tool also supports the definition of inbuilt rules which force the user to establish traceability links. Use of these rules is optional.

On the other hand, the configuration management tool was only used by the software subprojects. Using the configuration management tool for change management would restrict the use of the tool only to the software subprojects. Moreover, a large amount of the change requests had to be shared by several subprojects. This could result in problems in distributing information between

subprojects and projects. In addition, the new configuration management tool had not gained global acceptance in all sites of the company, and therefore there was a risk that all the software projects would not use the same configuration management tool. On the other hand, Lotus Notes was used by all technology groups in the company.

Several possibilities were examined to find a suitable tool environment for implementing new change management process support. The two strongest candidates were the new configuration management tool and Lotus Notes. The final choice was to implement support using the Lotus Notes system, because communication support was considered to be more valuable than traceability support.

Figure 31 illustrates the flow of actions between the change management tool, which was implemented to support the processes modelled, and the configuration management tool. The change request and its life cycle is stored into the Lotus Notes database by the change management system. The change request is linked to the software documents managed by the configuration management system using a Change Notice. When a change request is created, a Change Notice is automatically generated into the configuration management tool. The Change Notice includes a reference to the original change request. After the change has been implemented and the affected documents are returned into the configuration management tool, they are linked with the Change Notice. The Change Notice includes the following information:

- A reference to the original change request.

- A list of documents affected by the change.

As soon as the affected documents have been linked with the Change Notice, the list of affected documents is also visible in the Change Request document.

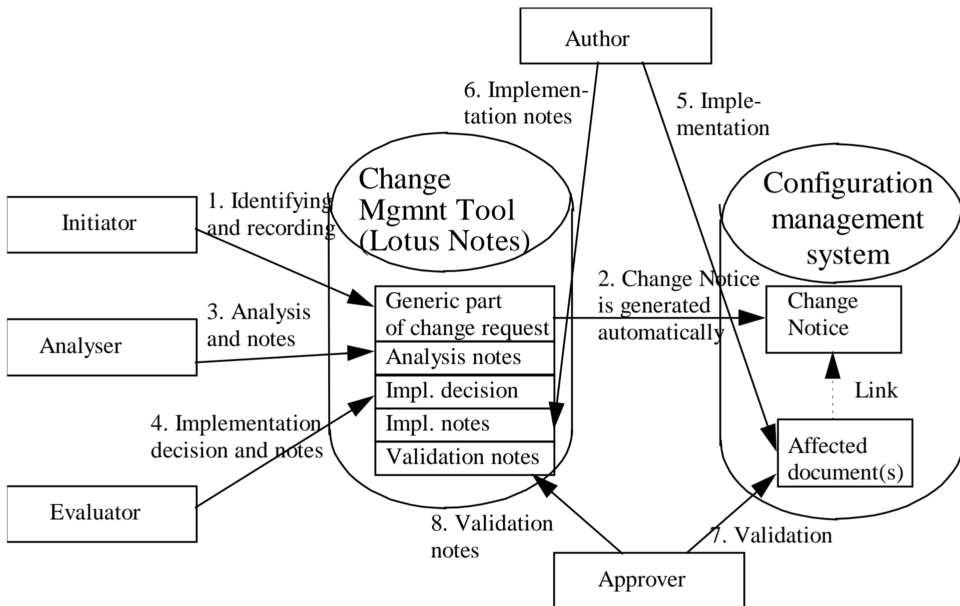


Figure 31. Integration of the change management tool and the version control tool.

7.10 Implementation and deployment of the defined solution

The change management tool implemented in the project contained process support for all the change management processes described in the previous sub-chapters. The deployment of new work processes was done simultaneously with the introduction of the new change management tool. The actual tool implementation was done by a separate implementation project, which was controlled by the project which defined the change management processes.

The next step after the definition of the processes was to model the data to be managed by the change management system. The data modelling was done using a process-oriented approach: the data needed and produced by each process phase was modelled separately. The data models were then exported into the

forms of the change management database. The change management system was then able to provide the user with the data fields needed in the particular phase the change document was in at the time. For example, the user was not requested to enter information related to the validation of the modification, if the change document was in the identification phase of its life cycle.

The communication between the individuals was supported by the change management system using two main mechanisms: the customised views to the change documents and connection to the E-mail system. The customised views to the change documents were defined to allow the users to browse change documents relevant to them. For example, the user could see only the change documents he or she was responsible for. The use of E-mail to inform people about the change documents, their statuses and the responsibilities related to them was modelled separately. Two basic kinds of strategies of sending E-mail were defined: automatic and manual sending. The automatic sending sends the E-mail to the defined persons automatically in predefined process phases. The manual sending procedure provides the opportunity to send an E-mail in certain predefined process phases, but waits for the user to trigger it and to define who the E-mail should be sent to.

The implementation strategy was simple: the goal was to implement the first version of the system as soon as possible for trial use. For that reason, the requirements of the system were elicited from a rather restricted group, i.e. the software development projects in two sites of the globally distributed company. The time used from the change management problem analysis to the release of the first version of the system was about half an year. The date of the first release was December 21, 1995. Seven revisions were done to the first release (last revision was made on April 25, 1996). All the revisions consisted of corrections to the implementation details or additions of input data fields.

The immediate advantages of the new solution were evaluated by the group involved in the definition of the new solution. These observations are based on the experiences from the pilot phase. The following advantages were identified:

- Support for measurement and analysis activities. The new solution included standard analysis and summary reports, and their use was enforced in the projects. The standard measurement introduced two clear advantages: (1)

projects could use the change information for estimation purposes, and comparison of measures across projects became possible, and (2) the data entered by the users was taken into use effectively, which increased the reliability of the data since people became motivated into entering it.

- Simple user interface. The user interface of the old error reporting tool was difficult and clumsy to use even for software engineers, and almost impossible for other people, such as marketing personnel or hardware designers. The graphical user interface of the new system was a huge improvement, which resulted in a good coverage of the change requests, since it was very simple to enter them into the system.

- Support for internal communication in the project and external communication between projects. Support for communication was provided by the system. In addition,the simple user interface improved information sharing, since the information was easily available and in a format which was easy to read, comprehend and utilise.

The system was originally designed for one product development site in a geographically distributed company, but the new change management system was quickly taken into use by other sites, too. This was of course enforced by the fact that the product projects were geographically distributed, and using a common tool for change management within a project is almost a necessity. Additionally, although the system was clearly aimed for the software development projects, it was also adopted by other technology groups, such as groups designing mechanic parts, marketing, etc.

The second version of the system was released on August 23, 1996. The main difference to the first version was the added support for change management related documentation. For example, the feature for creating and distributing review agendas was added to the system, as well as support for test reporting. Some reporting features were improved as well. The defined change management processes remained the same, only some related documentation processes were added to the system. By adding support to the documentation processes the integration of the change processes to the related documentation processes was improved. For example, when the test documentation was managed through the change management tool, the error documents recorded using the change man-

agement tool could be linked directly with the testing documentation. The last revision of the second system release was made on August 15, 1997.

The third version of the system was released in September 1997. The major difference in the structure of the system was to separate the error management part with related documentation support (test planning and reporting) into a separate system with its own user interface and database. This was done mainly because of the huge amounts of data stored into the error database. Modification needs also arose, for the following reasons:

- Redesigning the notification mail sending procedures, such as redefining when and how to send notification mails, and integration with other mail systems (initially only the internal Lotus Notes E-mail system was supported),

- Validation of input values was still a problem. For example, more automated input value checking was needed, and more default values and selection lists for input fields were requested,

- Compatibility with the Lotus Notes application development guidebook. Since Lotus Notes was used widely in the company, a guidebook defining a common look for all Lotus Notes applications was established. Since the change management system was designed before the guidebook, its design did not follow the guidelines defined in the Lotus Notes application development guidebook.

- New views into the database and new data fields for the change documents were needed.

The next release of the system is now (first half of year 1999) in the specification phase. The fourth release will include some additions to the current processes, and will also further expand the scope of the system. For example, the open item management process will be enhanced by expanding the risk management component. The system will provide better support for evaluating and managing risks related to open items before, during and after their implementation. Also, support for the customer feedback process will be included in the

system. This process is not exactly a software change process, but it will generate inputs to software change management processes.

Estimating the amount of users and projects using the defined change management system is very difficult, because the deployment has not been controlled and the projects have adopted the new system voluntarily. According to the person responsible for the tool in the organisation, several hundreds of projects have used the defined change management system during the period from 1996 to 1998. Currently (beginning of 1999) more than 150 projects or other similar entities have an active change management database in the change management system.[2] The person responsible for the change management tool assumes that all software projects use the defined change management system, and in addition several other technology groups have also adopted it.

7.11 Summary

This chapter presented the instantiation of the presented change management process model and the generic change management processes in one case study. Firstly, the needs and requirements related to the change management were defined by describing the initial processes and their problems and strengths. The change management problem classification was used in presenting the identified problems. Secondly, the new target state for the selected change management processes, which aim at responding to the problems identified in the initial status, was described. The target state was derived by tailoring the generic change management process descriptions characterised by the proposed process model to the special needs and requirements identified in the case study. Thirdly, the selected implementation solution was described and the history of the deployment of the system was briefly summarised.

---

[2] Exact number of projects cannot be published due to confidentiality reasons.

8 Evaluation of results

8.1 Introduction

This chapter examines the experiences of using the change management problem classification and process model in implementing a new change management system for the organisation described in chapter 7. The experiences are evaluated from three viewpoints: (1) problem analysis, (2) process definition, and (3) implementation. Experiences from the problem analysis made in the case organisation are used to evaluate the change management problem classification proposed by the thesis. The proposed process model is evaluated by summarising the experiences of defining the change management processes in the case study using the process model. Finally, the experiences from the implementation and deployment of the new change management system are summarised.

8.2 Evaluation of the change management problem classification

The proposed problem classification was found to be suitable for identifying the problem areas in change management and the requirements for the improvement of change management practices in the case study presented in chapter 7.

The effectiveness problems found were related to the effectiveness of the review practices. Because of the ineffectiveness of the reviews, the change needs were not identified as early as they could be, and the later they were recognised and managed, the more expensive and difficult the implementation was. The effectiveness problems required actions mostly in the procedures generating change requests, not directly in the change management procedures.

Communication problems were found frequently in the case study. Supporting both internal communication in the subproject, and communication between the subprojects and between the product projects was clearly a topic which had to be considered and supported at the level of change management processes and tools. Also, the informal sources of change requests were considered to cause

problems in projects. Improvements in communication problems could often be achieved directly by improving change management practices, for example by establishing procedures for managing informal sources of change requests or by better definition of roles and responsibilities in modification tasks.

The analysis and location problems found in the case study dealt with the difficulty of classifying multifaceted types of changes using the same classification, and with predicting the type of the change in the early phases of the change management process.

The problems concerning traceability issues in the case study were mostly related to the absence of a traceability link between the change and the software items modified because of the change. Also the lack of traceability support between the different software abstraction levels, e.g. the specification and design documents, resulted in problems in keeping them consistent with each other. The technical hindrance to establishing better traceability support was the incompatibility of the software configuration management environment and the change management system.

Actual decision-making problems were not found in the case organisation. Some requirements related to decision-making could be identified in the communication requirements, such as the problem of providing enough analysis information in the right format for the decision makers. Since decision-making and communication problems tend to be very tightly related with each other, these problem classes could be combined into one problem class.

The tool problems in the case study were mostly related to unpractical features in the current change management tools, or to the lack of support for tasks which could be easily automated but currently require lots of manual work. The tool problems are often related to other problem classes, for example when a problem with a tool makes communication or decision-making difficult.

Chapter 8.4 evaluates in detail how these problem areas were supported by the new, improved change management practices.

8.3 Evaluation of the change management process model

The thesis proposes a model for change management processes, which adapts to different types of changes. The model has been derived from the models presented in the literature, and it has been complemented with the experiences from the case studies. The value of the proposed model has been proven in this thesis by presenting a practical implementation of the model in a real world software development environment. The experiences of applying the model were good, and the defined change management solutions are still in active use.

Unlike most other change process models, the model proposed here explicitly supports different types of changes. This makes change management more flexible, since the variation of changes with respect to the amount of work needed, criticality of the modification and schedule varies a lot between changes.

The proposed model also identifies several change levels. The most distinct levels identified in the case organisations are product and project change levels. Supporting both product and project level change processes is particularly important when the product development is implemented in iterative development cycles, in which a new project is based on the baseline of an old product and a set of modification requests to be implemented in a new product. According to the experience of the writer of this thesis, this form of product development is nowadays very common in the development of embedded systems.

The change management process model and the related process descriptions proposed in the thesis are very generic in nature. The specific features of embedded software development mostly affect the requirements for the level of supporting the technical modification work (for example, regression testing). The process-related issues proved to be rather generic. For example, although communication support between subprojects is especially important in the development of embedded software, it can also be recognised in developing other types of software systems, as well.

As can be seen in the version history of the change management system briefly summarised in chapter 7.10, no change pressures to the original process defini-

tions emerged. The change needs were mostly related to the implementation details and to expanding the support provided by the system to the documentation processes closely interacting with the change management processes. The originally defined change management process types proved to fit the needs of the organisation extremely well.

Interviews of the end users were conducted to identify how the instantiation of the process model presented in chapter 7 succeeded in solving the specific change management problems identified in the case study (the problems are listed in chapter 7.7) Table 29 summarises the interview results. The first column lists the problems, and the second column evaluates the effect of the instantiation on the change management problems. The effect is evaluated using the following scale:

- **Worsened**. This grade was used if the interviewee thought the problem had become worse because of the new change management solution.

- **No effect**. This grade was used if the solution had no effect on the stated problem.

- **Improvement**. If the interviewee thought that there had been an improvement in the problem in question, this grade was used.

Table 29. Effects of the new implementation on the problems of the case organisation.

Problem	**Effect**
1. Ineffectiveness of the technical reviews	No effect
2. Lack of training in change management practices	Improvement
3. Informal sources of change request	Improvement
4. Lack of sharing of change requests between projects and subprojects	Improvement
5. Impracticality of using the severity classification for UI projects	Improvement
6. Modification requests in wrong lists	Improvement
7. Postponed change requests were lost	No effect
8. No feedback loop	Improvement
9. Poor traceability between changes and software items	Improvement
10. No link between change record and modified items	No effect
11. Inconsistencies between abstraction levels	No effect
12. No support for the generation of a change history list for the product review authorities	N/A
13. Wrong input values	Improvement
14. Input fields which are not used anywhere	No effect

The first problem, ineffectiveness of technical reviews, was considered to be unaffected by the new change management solution. The interviewees thought that the problem was dependent on the individuals working in the project, and could not be dealt with using process or tool solutions. The ineffectiveness was considered to be mostly caused by the lack of time to prepare oneself to a review session because of other work duties.

There was significant improvement in the second problem, lack of training courses and training material. In the opinion of the interviewees there were plenty of training courses and training material available on the change man-

agement processes and tools. The problem was more how to motivate people to attend the training, when they are under the pressure of tight project schedules, and how the people can find the useful manuals and training material from among the vast amount of material provided.

The interviewees identified a very strong improvement in the third problem, informal change request sources. They thought that the new processes and tools were used extensively to report change requests which previously had been processed through informal channels. This was considered to be the most positive aspect of the introduction of the new change management solution.

Some improvement was identified in the fourth problem, lack of sharing change requests between projects, but the subject was still considered to be problematic. However, the interviewees thought that this problem cannot be completely solved by any means, since analysing the effects of a change is inherently difficult in real life because of the complexity of the systems.

The way of managing severity classifications in the new system was considered to be practical and the interviewees could not identify new problems in using it, so problem 5 has been solved by the new solution. The interviewees thought that assigning a severity value to a change is always person-dependent and sometimes not clear, but the current support for managing severity classifications was considered to be feasible.

Improvements were identified in the management and processing of the changes which are recorded in the wrong list. However, the interviewees said that the change requests were still recorded in the wrong lists, and the improvements have mostly been in the processing of those changes. This was considered to be sufficient enough, since by the time the change request is created its root causes cannot yet be well known, and it is natural that it can be on a wrong list until deeper analysis has been done.

The postponed change requests were still often lost according to the interviewees. The new solution tried to tackle the problem by asking the user to define which project should take care of the postponed change request instead of just postponing it, as was done in the old solution. However, the interviewees thought that this procedure was not effective, since at the time the change re-

quest is postponed the people doing it do not know which future projects will be able to deal with it.

The feedback loop from the change data to new projects was slightly improved according to the interviews. The types of change requests were analysed and this information was used in planning tasks in other projects.

Traceability between changes and software items was considered to have been slightly improved. The traceability still relies on manual recordings made by the software designers and is still prone to human error. However, the recordings are made more often and with greater care than they were done before.

The automated linking facility (see the concept of change notice on page 155) between the configuration management and change management tools was not used by the projects in which the interviewed people were involved.

Inconsistencies between abstraction levels were still frequent in the projects, and no effect was identified by the interviewees. The higher level documentation was usually left unchanged after its acceptance, and the changes were done only to the lower level documentation.

Problem 12 was not relevant to the interviewees, since the change history lists were not required anymore.

Pproblem 13, wrong input values, was drastically improved according to the interviewees. The new tool support provided clear guidance and automatic checking of the input values, both advicing the user in entering the right information and preventing the user from entering false values.

The forms used for recording change requests and change related information were still considered to include fields which were not used anywhere. The interviewees assumed that this was usually caused by the fact that the same forms were used by different types of projects, and some fields were project-specific.

8.4 Experiences from the implementation and deployment

The deployment of the new system was very successful, when measured by the great speed that the projects took it in use, and how widely the system was adopted in the company. The software projects very quickly adopted the new system with the accompanying process guidelines, templates and tool support. The new change management practices with the support for the new change management system were adopted not only by the software projects on the site where the system was implemented, but also by other development sites of the company and other technology groups besides the software projects. Also, the system was taken into use not only by development projects, but also by line organisations and other similar instances, for supporting meetings, etc.

The following table (Table 30) summarises how the new change management solution responds to the problems identified in the initial state of the change management.

Table 30. Evaluation of the solution against the problem areas.

Problem	**Evaluation of the solution**
Effectiveness	• Better distribution of changes across projects • Improved analysis functionalities for managing duplicates, misunderstandings, etc.
Communication	• Improved communication across projects and technology groups • Better visibility of change information
Analysis and location	• Support for analysis and location through enhanced communication • Enhanced support for documentation
Traceability	• Automated change history for change documents • Support for recording traceability links between the change document and affected software items
Decision making	• Better documentation of changes • Better communication of change information
Tool-related problems	• Better user interface • Unified tool environment

The new change management solution has had an impact on the effectiveness of the change processes with respect to two main issues. Firstly, the channels for distributing change information across projects have been improved, enabling the projects to react earlier to the changes initiated by other projects. Secondly, the improved analysis functionalities have improved change request management, i.e. the analysis and removal of duplicate change requests, change requests caused by misunderstandings, etc.

The fact that the new change management solution has been taken in use not only by the software groups, but also by other technology groups, has resulted in

a big improvement in the communication of changes between technology groups. The communication between software projects has been also improved by the enhanced change information distribution mechanisms. Although from the project viewpoint the communication within the software project group did not change, the improved tool support provided better visibility of the change information for individual software engineers. The communication from the project level to the project management level has been improved by the better reporting support. Common report templates have been defined to report change-related information to the project management.

The analysis and location problems have mostly been affected through improved communication, traceability and documentation. The improved communication has enhanced the change request analysis and location of the product parts to be modified, especially when several projects or technology groups are involved in the change activity. Also, the documentation of the results of the analysis phase has been improved, including better support for recording traceability links between the change document and the software items affected by the change, and for the recording of the implementation proposals.

The traceability of individual changes has been improved by the tool environment, which automatically provided a change history for individual change requests. Also, the tracking activities have been supported by the customised views for monitoring change requests. The tool environment has also provided support for recording traceability links between the change document and the software items affected by the change.

The decision-making problems were not found to play a critical role in the case organisation in the initial problem analysis phase. Therefore, no special effort was directed to support the decision-making process. The positive effects of the new solution to the decision-making have been achieved through the better documentation of changes and analysis information, and better communication of change-related information.

The two main tool-related improvements have been the improved user interface of the change management tool (compared with the former set of change management tools used) and the unified tool environment. The improved user interface includes several components. The graphical user interface provides support

for the user in entering the change information, e.g. selection lists, context sensitive help, etc. It also supports the user in browsing changes and finding relevant information from the change database. The database solution has provided better means for browsing the change data and making common and customised reports from the change data, when compared to the old text editor or word processor based documents the projects previously used for storing change information. The unified tool environment has provided significant improvements mainly relating to the communication aspect, when compared to the earlier situation, where the tools and processes used by different projects and technology groups were totally different and separate. The unified tool environment has also directed the projects and technology groups to use common processes for managing changes.

The implementation is evaluated here only on a qualitative basis, since a quantitative evaluation was not feasible for the following reasons:

- Quantitative comparison of the situation before the new system and after its deployment proved to be difficult, because reliable and meaningful measurement data was impossible to derive from the old change management system. The old system was text-based, which resulted in a great variety of input values. This resulted in difficulties in deriving summaries from the change data. For example, the effort used for analysing the change request was expressed in hours for some changes, months for others, and qualitatively, e.g. 'only a short time', for others.

- Due toconfidentiality reasons, the company did not allow any product-related information (exact numbers of projects, change requests or information about product components) to be presented in this thesis.

Perhaps the best indication that the new system provides improvements over the old change management practices is the fact that the system was so rapidly and widely adopted in the company. The development projects were completely free to choose any environment for change management, including the possibility to continue using the old systems. Nevertheless, the software projects unanimously chose the new system, and it was soon adopted by other projects and organisational entities, as well.

Due to the rapid propagation of the system, the support and maintenance of the system itself became demanding. The original requirements of the system were derived from the software projects in two sites of the company, and suddenly the system was used worldwide by a wide variety of technology groups and organisational entities. Requirements arose for tailoring the process guidelines, templates and tool-supported forms to the special needs of the different types of users. The support and maintenance of the change management system itself addressed the change management problems.

8.5 Summary

This chapter evaluated and summarised the experiences from the case study. The presented process model provided a solid basis for defining the change management support in the organisation. It provided a good coverage of the software change processes, and provided a clear classification of the different types of change processes. However, even if the process model was based on an extensive study of actual change processes related to the development of embedded software, and the requirements and problems especially in that domain were evaluated, the change management process model and the related process descriptions are very generic in nature. The special features and problems in the development of embedded software had the biggest impact on the technical level of the modification, and they also put more emphasis on certain features on the process support level.

9 Conclusions

9.1 Answers to research questions

The thesis aimed at increasing our knowledge about software change management. The research questions were the following:

- What are the essential change management problems and improvement requirements found in the development of embedded software?
- What kind of management process would aid in responding to these requirements?
- How can the proposed processes be implemented and enacted in practice?

The first part of the thesis presented a summary of the literature and change management related research. Change management has traditionally been studied in the context of software maintenance research. Recently, as evolutive software processes have become more popular, it has been studied as a component of iterative development cycles and in connection with software configuration management. This thesis provides a summary of change management related research.

The literature-based theoretical viewpoint to software change management was complemented with three background case studies, which aimed at presenting insights into state-of-the-practice software change management. The thesis provided a summary of the three case studies from the viewpoint of their special characteristics in software management.

These two sources, an analysis of software change management related research and three background case studies, provided the answers to the first research question. By combining these two research methods, both the theoretical and practical viewpoints could be covered. As a result, a classification of typical problems and improvement requirements for change management were defined. The problems were examined especially from the viewpoint of the development

of embedded software by relating the problems to the special features of the development of embedded software.

The second research question was answered by presenting a model of software change management processes and related generic process descriptions in chapter 5.

The process model and the related generic process descriptions were used in designing and implementing new change management practices in one case organisation. The implementation was presented in the chapter 7. The case implementation part of the thesis provides answers to research question three.

The results of the design science research can be assessed against the criteria of value. (March & Smith 1995) The models constructed in this thesis were primarily assessed by examining if they work and are used in the company examined in the case study. Some qualitative indications of whether the new system introduced improvements over the old change management practices were also provided.

9.2 Generalisation of the results

The title of the thesis defines the scope of the present study to be development of embedded software. The change management related literature was studied in a wider context, i.e. not limiting it only to change management in the context of embedded software. The case studies were performed in organisations developing products with embedded software. The thesis provides a characterisation of software change management from both theoretical and practical viewpoints, and the requirement classification part also examines the special problems and requirements in the change management of embedded software. The process model and generic process descriptions presented have been derived and implemented in the domain of embedded software, thus no analysis of their suitability or unsuitability in other domains was provided. This thesis aims at describing the model and its validation and implementation examples in the degree, which allows the reader to judge whether the context of the model is sufficiently analogous to the case it is intended to apply to, and to what extent the results are

relevant. Research can be generalised according to the following forms of generalizability (Kvale 1996):

- Naturalistic generalisation, which rests on personal experience, which develops as the person gets experienced in the subject area. Naturalistic generalisation derives from tacit knowledge of how things are.

- Statistical generalisation, which is formal and explicit, and always bases on subjects selected at random from a population. Random selection of interviewees and quantified interview results allow statistical generalisation of the results.

- Analytical generalisation, which involves judgement about the generalisation based on an analysis of the similarities and differences of the two situations.

The generalisations made in this thesis are based on naturalistic generalisation based on the theoretical knowledge and empirical experience of the author on change management, and on the intensive co-operation with the case organisations in several improvement projects dealing with change management. In the author's view the change management process model and the related generic process descriptions are generic enough to be used in other domains.

9.3 Future research

This thesis has presented an analysis of change management practices in four organisations, but implemented a new change management solution for only one of the organisations studied. The organisation in which the new change management solution has been developed is a large company. Further studies are needed to examine how a formalised generic change management process can be tailored to meet the needs of a particular organisation, to avoid unnecessary bureaucracy in different types of development organisations.

The generic change process and its instantiations generated in this study are an example of a waterfall-type process, where the main vehicle for managing changes within the process flow is the iteration between the process phases. Some of the changes, such as straightforward error situations, can be effectively

processed using this type of process model. However, there are change situations which do not naturally follow the waterfall-type process model. For example, a vague improvement proposal may be more effectively managed by using a change process which makes more use of an iterative approach to searching the causes for the change and generating solution proposals.

The case studies presented in this thesis used interviews as the main vehicle for studying the work practices in the organisations studied. The experiences indicate that advanced methods are needed to understand the actual work flows and practices in software development and maintenance. The interviews proved to be well suited to finding out the problems and improvement requirements for the processes, but eliciting process and workflow information using the interviewing method proved to be problematic.

The results were evaluated using qualitative research methods. Making a reliable quantitative evaluation proved to be difficult, if not impossible. In future research dealing with similar kinds of process improvement issues, the quantitative evaluation of the improvement results should be planned more carefully from the very beginning of the research.

One trend in software development is towards building systems using reusable software components. Therefore, managing software changes in environments where the software modules are shared by several products will probably gain importance in the future. More research is needed in exploring the requirements for change management in reuse-based software development and in constructing change management environments to support this type of software projects.

10 Epilogue

The software engineering community aims at providing effective methods and tools for solving problems and improving software development work in practice. In order to do that, it is necessary to learn more about the software development work and what the actual problems are. The four case studies summarised in this thesis provide an insight into the requirements of change management processes and the problems in practical software development.

One of the case studies presented in this thesis was continued to include the definition, implementation and deployment of the new, improved process support for the selected process area. The presented case study provides an example of how to proceed in defining and introducing new solutions for software development work. The evaluation of the success of the new process environment turned out to be problematic, since the effects of the process changes on the performance of the software development project are difficult to prove. The comparison of the situation before and after the deployment of the new processes was troublesome. The data available from the initial situation was not statistically reliable or valid, partly because of the informality of the change management processes, and partly because of problems and inconsistencies in tool support. In addition, the vast amount of different variables affecting the quality and performance of the software development work makes it extremely difficult to separate the effects of improvement actions from other affecting factors. Nevertheless, a qualitative evaluation of the work could be carried out by discussing with the people involved in the implementation of the new change management system and with the end users of the system.

The thesis presented a snapshot of the problems and characteristics of software change processes in four organisations developing embedded software. It derived a descriptive classification of the different types of problems faced in change management, and a generic model of the change processes identified in the organisations. The model provided unique flexibility in adapting the processes with respect to

- different types of changes, and
- different levels of change processes.

The model does not restrict itself to any specific phase of the software life cycle, but can be used within all software development and maintenance models.

References

Abbattista, F., Lanubile, F., Mastelloni, G. & Visaggio, G. 1994. An experiment on the effect of design recording on impact analysis. Proceedings of International Conference on Software Maintenance. Pp. 253 – 259.

Ackoff, R. 1967. Management misinformation systems. Management Science. (Vol. 14, No. 4). Pp. 147 – 156.

AMES Consortium. 1993. AMES technical annex, Version 1.0. ESPRIT Project 8156.

Arnold, R. 1993. Software impact analysis. Tutorial Notes, IEEE Conference on Software Maintenance. Montreal, Canada.

Arnold, R. & Shawn, A. 1996. Software change impact analysis. IEEE Computer Society Press. 392 p. ISBN 0-8186-7384-2.

Bandinelli, S., Fugetta, A. & Ghezzi, C. 1995. Software process model evolution in the SPADE environment. IEEE Transactions on Software Engineering. Pp. 1128 – 1144. (no 12).

Barros, S., Bodhuin, T, Escudie, A, Queille, J. & Voidrot, J. 1995. Supporting impact analysis: a semi automated technique and associated tool. IEEE Software. IEEE. Pp. 42 – 51.

Barton Cunningham, J. 1997. Case study principles for different types of cases. Quality & Quantity. Kluwer Academic Publishers. Pp. 401 – 423.

Benington, H. 1956. Production of Large Computer Programs. Proceedings of the ONR Symposium on Advanced Program Methods for Digital Computers. Pp. 15 – 27.

Bennet, K. 1993. An overview of maintenance and reverse engineering. Chapter 2. In: Henk van Zuylen (ed.) The REDO Compendium: Reverse Engineering for Software Maintenance. John Wiley & Sons. Pp. 13 – 34.

Bennet, K. 1996. Software evolution: past, present and future. Information and software technology. Elsevier Science B.V. Pp. 673 – 680.

Bergey, J., Clements, P, Cohen, S., Donohoe, P., Jones, L., Krut, B., Northrop, L., Tilley, S., Smith, D. & Withey, J. 1998. DoD product line practice workshop report. SEI. 59 p. (CMU/SEI-98-TR-007)

Berns, G. 1984. Assessing software maintainability. Communications of ACM. ACM. (Vol. 27, no 1.) Pp. 14 – 23.

Bersoff, E. & Davis, A. 1991. Impacts of life cycle models on software configuration management Communications of the ACM. (Vol. 34. No. 8) Pp. 104 – 118.

Bjerknes, G., Bratteteig, T. & Espeseth, T. 1991. Evolution of finished computer systems. The dilemma of enhancement. Scandinavian journal of information systems. (Vol. 3) Pp. 25 – 45.

Boehm, B. 1976. Software engineering. IEEE Transactions on Computers. (Vol. C-25, No. 12) Pp. 1226 – 1241.

Boehm, B. 1988. A spiral model of software development and enhancement. Computer. (Vol. 21.) Pp. 61 – 72. ISSN 0018-9162

Boldyreff, C., Burd, E. & Hather, R. 1994. An evaluation of the state of the art for application management. Proceedings of the International Conference on Software Maintenance. IEEE Computer Press. Pp. 161 – 169.

Briand, L., Basili, V., Kim, Y. & Squier, D. 1994. A change analysis process to characterize software maintenance projects. Proceedings of the International Conference on Software Maintenance. IEEE. Pp. 38 – 49.

Brown, A. 1993. Specifications and reverse-engineering. Journal of Software Maintenance and Research. John Wiley & Sons. (Vol. 5). Pp. 147 – 153.

Brown, A., Christie, A. & Dart, S. 1995. An examination of software maintenance practices in a U.S. government organization. Software maintenance: Research and practice. John Wiley & Sons. (Vol. 7). Pp. 223 – 238.

Canning, R. 1972. The maintenance iceberg. EDP Analyzer. (Vol. 10, no 10.) Pp. 1 – 4. ISSN 0012-7523

Capretz, M. & Munro, M. 1994. Software configuration management issues in the maintenance of existing systems. Software maintenance: Research and practice. John Wiley & Sons. (Vol. 6). Pp. 1 – 14.

Case material. Internal documentation of the organisation examined in the case study. Exact reference to a document cannot be made due to confidentiality reasons.

Chapin, N. 1993. Behind and ahead – A decade's perspective. Proceedings of the conference on software maintenance. IEEE. Pp. 411 – 412.

Chen, S., Heisler, K., Tsai, W., Chen, X. & Leung, E. 1990. A model for assembly program maintenance. Software maintenance: Research and practice. John Wiley & Sons. (Vol. 2). Pp. 3 – 32.

Chikofsky, E. & Cross, J. 1990. Reverse engineering and design recovery: A taxonomy. IEEE Software. (Vol. 7, No 1). Pp. 13 – 17.

Cutillo, F., Lanubile, F. & Visaggio, G. 1996. Managing a software system and keeping it internally consistent during its evolution. Proceedings of the 8th International Conference on Software Engineering and Knowledge Engineering. KSI Publications. ISBN 0-9641699-3-2

Dart, S. 1991. Concepts in configuration management systems. Proceedings of the 3rd International Workshop on Software Configuration Management. Baltimore, Maryland: ACM Press. Pp. 1 – 18.

Edelstein, D. & Mamone, S. 1992. A standard for software maintenance, A framework for managing and executing software maintenance activities. Computer. Pp. 82 – 83. (June)

Eisenhardt, K. 1989. Building theories from case study research. Academy of Management Review. (Vol 14, No 4). Pp. 532 – 550.

ESA. 1991. PSS-05-0 issue 2. ESA Software Engineering Standards, Issue 2. European Space Agency. 130 p.

Forte, G. 1993. Software configuration management. CASE outlook. (Vol. 7.) ISBN 0895-2108

van Genuchten, M., Brethouwer, G., van den Boomen, T. & Heemstra, F. 1992. Empirical study of software maintenance. Information and Software Technology. Butteworth-Heineman Ltd. (Vol. 34, No. 8). Pp. 507 – 512.

Glaser, B. & Straus, A. 1967. The discovery of grounded theory: Strategies for qualitative research. New York: Aldine Publishing Company.

Goguen, J. & Linde, C. 1997. Techniques for requirements elicitation. In: Thayer, R. & Dorfman, M. Software requirements engineering. 2nd ed. Los Alamitos, USA: IEEE Computer Sociaty Press. 483 p. ISBN 0-8186-7738-4

Gresse, C., Hoisl, B. & Würst, J. 1995. A process model for GQM-based measurement. Software-Technologie-Transfer-Initiative Kaiserslautern. 229 p.

Hather, R., Burd, E. & Boldyreff, C. 1995. A quality assessment method for application management. Proceedings of Software Quality Management `95. Pp. 299 – 310.

Harjani, D.-R. & Queille, J.-P. 1992. A process model for the maintenance of large space systems software. Proceedings of Conference on Software Maintenance. Los Alamitos: IEEE Computer press. Pp. 127 – 136. ISBN 0-8186-2980-0

Horn, R. 1992. Participant's manual for developing procedures, policies & documentation. Massachusetts: Information Mapping, Inc. ISBN 0-912864-05-2

Humphrey, W. 1989. Managing the software process. Addison Wesley Publishing Company, Inc. 494 p. ISBN 0-201-18095-2

IEEE 828. 1990. IEEE standard for software configuration management plans (ANSI). IEEE. 16 p.

IEEE 1993. IEEE Std 1219. IEEE standard for software maintenance. IEEE Standards Collection. IEEE. 39 p. ISBN 1-552433-442-X

Ince, D. 1994. Introduction to software quality assurance and its implementation. McGraw-Hill. 202 p. ISBN 0-07-707924-8

Isomursu, P. 1995. A software engineering approach to the development of fuzzy control systems. Espoo: VTT. 79 p. + app. 55 p. (VTT Publications 230) ISBN 951-38-4768-3

Jones, C. 1989. Software enhancement modelling. Software maintenance: Research and Practice. John Wiley & Sons. (Vol. 1). Pp. 91 – 100.

Jones, C. 1996. Strategies for managing requirements creep. Computer. (June 1996) Pp. 92 – 94.

Karjalainen, J., Mäkäräinen. M., Komi-Sirviö, S. & Seppänen, V. 1996. Practical process improvement for embedded real-time software. Quality Engineering. (Vol. 8, No. 4.) Pp. 565 – 573.

Kellner, M. 1993. Ten years of software maintenance: progress or promises? Proceedings of the international conference on software maintenance. IEEE. Pp. 406 – 408.

Ketabchi, M. & Sadeghi, K. 1996. Applying object technology to software analysis and maintenance system development. Journal of Systems and Software. Elsevier Science. (Vol. 32). Pp. 41 – 56.

Kvale, S. 1996. Interviews: An introduction to qualitative research interviewing. SAGE Publications. 326 p. ISBN 0-8039-5820-X

Lam, W., Loomes, M. & Shankararaman, V. 1999. Managing requirements change using metrics and action planning. Proceedings of the Third European Conference on Software Maintenance and Reengineering. IEEE. Pp. 122 – 128. ISBN 0-7695-0090-0

Lano, K. & Haughton, H. 1993. Reverse engineering and software maintenance: A practical approach. London: McGraw Hill. 251 p. ISBN 0-07-707897-7

Layzell, P. & Macaulay, L. 1994. An investigation into software maintenance - perception and practices. Journal of Software Maintenance: Research and Practice. John Wiley & Sons. (Vol. 6.) Pp. 105 – 120.

Lehman, M. 1980. Programs, lifecycles and laws of software evolution. Proceedings of IEEE. IEEE. (Vol. 68, No. 9.) ISSN 0018-9219

Lehman, M. 1998. Software's future: Managing evolution. IEEE Software. IEEE. Pp. 40 – 44. (Jan-Feb)

Lientz, B. & Swanson, B. 1980. Software maintenance management. A study of the maintenance of computer application software in 487 data processing organizations. Addison Wesley. 213 p. ISBN 0-201-04205

March, S. & Smith, G. 1995. Design and natural science research on information technology. Decision support systems. Elsevier Science. Pp. 251 – 266.

McClure, C. 1989. The three Rs of software automation. Prentice Hall. 304 p. ISBN 0-13-915240-7

de Michelis, G., Dubois, E., Jarke, M. Matthes, F., Mylopoulos, J., Papazoglou, M., Pohl, K., Schmidt, J., Woo, C. & Yu, E. 1997. Cooperative information systems: A manifesto. In: Papazoglou, M. & Schlageter, G. Cooperative information system: Trends and directions.

Mittag, L. 1996. Trends in hardware/software dodesign. Embedded Systems Programming. (Vol. 9, No. 1.) Pp. 36 – 45.

Mäkäräinen, M. 1996. Application management requirements for embedded software. Espoo: VTT. 99 p. (VTT Publications 286). ISBN 951-38-4944-9

Mäkäräinen, M. & Komi-Sirviö, S. 1996. Practical process improvement for embedded real-time software. Proceeding of Fifth European Conference on Software Quality. Irish Quality Association. Pp. 408 – 416.

Nosek, J. & Palvia, P. 1990. Software maintenance management: changes in the last decade. Journal of Software Maintenance: Research and Practice. John Wiley & Sons. Pp. 157 – 174. (Issue 2)

Ojennes, D. 1998. Debugging embedded system software. Embedded system engineering. Pp. 36 – 40. (May)

Olsen, N. 1993. The software rush hour. IEEE Software Magazine. IEEE. Pp. 29 – 37. (September)

Osborne, W. & Chikofsky, E. 1990. Fitting pieces to the maintenance puzzle. IEEE Software Magazine. IEEE. Pp. 11 – 12. (January)

Paulk, M., Weber, C., Garcia, S., Chrissis, M. & Bush, M. 1993. Key practices of the capability maturity model, Version 1.1. SEI. 30 p. (CMU/SEI-93-TR-25)

Parnas, D. 1996. Mathematical description and specification of software. Lecture notes. Invited lecture at Fraunhofer IESE, Kaiserslautern, Germany. 2nd September 1996.

Pressman, R. 1992. Software engineering: a practitioner's approach. New York: McGraw-Hill Inc. 793 p. (3rd ed.) ISBN 0-07-050814-3

Pressman, R. 1995. Software according to Niccolò Machiavelli. IEEE Software. IEEE. Pp. 101 – 105. (January 1995)

Queille, J-P., Voidrot, J-F., Wilde, N. & Munro, M. 1994. The impact analysis task in software maintenance: A model and a case study. Proceedings of International Conference on Software Maintenance. IEEE. Pp. 234 – 242.

Rombach, H. 1987. A controlled experiment on the impact of software structure on maintainability. IEEE Transactions on Software Engineering. (Vol. SE-13 No. 3.) Pp. 344 – 354.

Royce, W. 1970. Managing the development of large software systems: concepts and techniques. WESCON Western Electronic Show and Convention. Los Angeles: WESCON. Pp. A/1-1 – A/1-9

Rubager, S., Ornburn, S. & LeBlanc, J. 1990. Recognizing design decisions in programs. IEEE Software. IEEE. Pp. 46 – 54. (January)

Schneidewind, N. 1987. The state of software maintenance. IEEE Transactions on software engineering. IEEE. (Vol. SE-13, No. 3.) Pp. 303 – 310.

Seppänen, V., Kähkönen, A.-M., Oivo, M., Perunka, H., Pulli, P. & Isomursu, P. 1997. Strategic needs and future trends of embedded software. Embedded System Engineering. (Vol. 5, No. 2.). Pp. 52 – 66.

van Solingen, R. & Berghout, E. 1999. The goal/question/metric method: a practical guide for quality improvement of software development. McGraw-Hill. 195 p. ISBN 007-709553-7.

Sommerville, I. 1982. Software Engineering - 2nd edition. Addison-Wesley publishing company. 334 p. ISBN 0-201-14229-5

Sneed, H. 1995. Planning the reengineering of legacy systems. IEEE Software. (Vol 12, No 1.)

Stark. G., Kern, L. & Vowell, C. 1994. A software metric set for program maintenance management. Journal of systems and software. Elsevier Science. (Vol. 24). Pp. 239 – 249.

Suitiala, R. 1993. Work-oriented development of interactive software tools. Understanding the work of software maintainer and making an interactive software tool for them. Espoo: VTT. 176 p. VTT Publications 139. ISBN 951-38-4257-6

Swanson, E. 1976. The dimensions of software maintenance. Proceedings of 2nd international conference on software engineering. San Francisco, USA. Pp. 492 – 497.

Taramaa, J. 1998. Practical development of software configuration management for embedded systems. Espoo, Finland: VTT, Technical Research Centre of Finland. 147 p. (VTT Publications 366). ISBN 951-38-5344

Tichy, W. 1988. Tools for Software Configuration Management. Internal Workshop on software Version and Configuration Control. Stuttgart, Germany: Teubner Verlag. Pp. 1 – 20.

Turski, W. 1981. Software stability. Proceedings of the 6th ACM European regional conference on systems architecture. Pp. 107 – 116.

Ward, M. 1993. Abstracting a Specification from Code. Journal of software maintenance: Research and Practice. John Wiley & Sons. (Vol. 5) Pp. 101 – 122.

Weiderman, N., Bergery, J., Smith, D. & Tilley, S. 1997. Approaches to Legacy System Evolution. SEI. (CMU/SEI-97-TR-014)

Vierimaa, M., Kaikkonen, T., Oivo, M. & Moberg, M. 1998. Experiences of practical process improvement. Embedded Systems Programming Europe. (Vol. 2, No. 13). Pp. 10 – 20.

Zahran, S. 1997. Software process improvement – practical guidelines for business success. Addison Wesley. 447 p. ISBN 0-201-17782

Published by

Vuorimiehentie 5, P.O.Box 2000, FIN-02044 VTT, Finland
Phone internat. +358 9 4561
Fax +358 9 456 4374

Series title, number and report code of publication

VTT Publications 416
VTT–PUBS–416

Author(s)
Mäkäräinen, Minna

Title

Software change management processes in the development of embedded software

Abstract

The goal of the research presented in this thesis is to examine software change management processes in order to identify essential change management problems and improvement requirements, to define processes which would aid in solving these problems, and give an example of how these processes can be implemented in practice.

The subjects of the empirical research part of the study have been four Finnish companies which develop embedded software. Therefore, the focus of the study is on the processes which are used in developing embedded software. However, the literature study explores the problems of software change management from a more generic viewpoint, and can also be used as a reference for software development done for other purposes.

Three of the four case studies are used in deriving the generic change manage-ment problem classes and process descriptions. These three case studies include only analysis of the change processes and problems related to them. The fourth case study is used for illustrating how the proposed problem classification can be used in change process analysis, and how the proposed process models can be instantiated in practice. The change processes were not only analysed, but the study also included the definition of new processes, the planning of their implementation, and the implementation and enactment of the new processes in the organisation.

Keywords
software change management, software configuration, software maintenance, process improvement, process modelling, process analysis

Activity unit
VTT Electronics, Embedded Software, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland

ISBN			Project number
951–38–5573–2 (soft back ed.) 951–38–5574–0 (URL: http://www.inf.vtt.fi/pdf/)			
Date July 2000	Language English	Pages 185 p. + app. 56 p.	Price E

Name of project	Commissioned by
Series title and ISSN VTT Publications 1235–0621 (soft back ed.) 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)	Sold by VTT Information Service P.O.Box 2000, FIN–02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374