

Configuration Management for Distributed Development – Practice and Needs

Ulf Asklund



Dissertation 10, 1999

Department of Computer Science
Lund Institute of Technology
Lund University

ISSN 1404-1219
Dissertation 10, 1999

Thesis submitted for partial fulfillment of
the degree of licentiate.

Department of Computer Science
Lund Institute of Technology
Lund University
Box 118
SE-221 00 Lund
Sweden

E-mail: Ulf.Asklund@cs.lth.se
WWW: <http://www.cs.lth.se/~ulf>

© 1999 Ulf Asklund

Abstract

Configuration management (CM) includes synchronizing and supporting developers in their common development and maintenance of a system. The rapid development of the Internet and time-to-market pressure affects software development. In order to utilize skilled personnel despite geographical location, groups of developers are now working all over the world on the development of common systems, a situation called distributed development. From different locations they may need to concurrently modify thousands of files, sometimes the same files, within the product. When the prerequisites change this implies new, and harder, demands on CM support.

This thesis presents two studies on configuration management. The first one focuses on a CM system and an evaluation of its functionality in the particular situation of a company with well-defined CM routines. Situations where the system turned out to be cumbersome to use or lacking functionality are identified and improvements are outlined for how the problems can be solved or the situations better supported.

The second study classifies different cases of distributed development and client-server architectures, identifies three development strategies and discusses the properties of different synchronization models with respect to distribution. The analysis is supported by interviews with several companies and is focused on CM aspects related to distributed development. Experiences and practical guidance for some CM key-areas are also presented.

The results presented in this thesis provide an understanding of concurrent and distributed development and their demands on CM and CM tools. It is concluded that the CM tools of today do not adequately support distributed groups, i.e. highly interactive distributed development – a problem that must be addressed by future research.

Acknowledgments

The research presented in this thesis was carried out within the Programming Environments Group at the Department of Computer Science, Lund University. I would like to thank my supervisor Boris Magnusson, the leader of the group, both for introducing me to the areas of configuration management and support for collaborative work, and for his support throughout my thesis work.

The studies described in the thesis could not have been performed without the help from a number of individuals and companies letting me interview them and their willingness to share their experiences. Special thanks to Krister Erlansson for many fruitful discussions and to Annita Persson and Göran Östlund for their confidence and engagement.

Thanks to former and present members of the Programming Environments Group. It is a pleasure to work with you all. Thank you Görel Hedin, Göran Fries, Roger Henriksson, Klas Nilsson, Torsten Olsson, Elizabeth Bjarnason, Anders Dellien, Daniel Einarsson, Patrik Persson, Anders Ive, and Mathias Haage.

This work has been financially supported by NUTEK, the Swedish National Board for Industrial and Technical Development and by VI, the association of Swedish Engineering Industries.

Contents

| | |
|---|----------|
| Introduction | 5 |
| 1 Research focus | 6 |
| 1.1 Research questions | 7 |
| 2 Research methods | 8 |
| 3 Summary of results | 8 |
| 3.1 Shortcomings of approaches to date | 10 |
| 4 Research results. | 11 |
| 4.1 Main contributions | 12 |
| 4.2 Summary of papers. | 13 |
| 4.3 Future research | 14 |
| References | 14 |

Paper I: A Case-Study of Configuration Management with ClearCase in an Industrial Environment 17

| | |
|---|----|
| 1 Introduction | 18 |
| 2 Background | 19 |
| 2.1 Introduction to ClearCase | 19 |
| 2.2 Kockums Computer Systems | 23 |
| 2.3 The COOP/Orm environment overview. | 24 |
| 3 CM overview at KCS | 25 |
| 3.1 The CM history in the company. | 25 |
| 3.2 How ClearCase is used today. | 26 |
| 4 Requirements and shortcomings of ClearCase | 26 |
| 4.1 The version selection problem for developers | 27 |
| 4.2 Support for short term sub-project | 28 |
| 4.3 Merging and integration | 29 |
| 4.4 Support for delivery and consistent configurations | 30 |
| 4.5 Support for group awareness. | 31 |
| 5 Relevant functionality in the COOP/Orm environment | 33 |

| | | |
|-----|---|----|
| 6 | Suggested improvements in ClearCase | 34 |
| 6.1 | Semantic checks for views | 34 |
| 6.2 | Relations among branches | 34 |
| 6.3 | Versions of directories | 35 |
| 6.4 | Awareness | 35 |
| 7 | Future work / further studies | 36 |
| 8 | Conclusions | 36 |
| | Acknowledgments | 37 |
| | References | 37 |

Paper II: Distributed Development and Configuration Management 39

| | | |
|-----|---|----|
| | Preface | 39 |
| | Summary | 40 |
| 1 | Introduction | 42 |
| 1.1 | Objectives | 43 |
| 1.2 | Scope | 44 |
| 1.3 | Reading instructions | 44 |
| 2 | Distributed development | 46 |
| 2.1 | Cases of distributed development | 47 |
| 3 | Configuration management (CM) | 53 |
| 3.1 | Strategies/working modes | 53 |
| 3.2 | CM from a management perspective | 55 |
| 3.3 | CM from a developmental perspective - tool support .. | 57 |
| 3.4 | Summary | 62 |
| 4 | Synchronization models | 64 |
| 4.1 | Checkout/checkin | 64 |
| 4.2 | Composition | 66 |
| 4.3 | Long transactions | 68 |
| 4.4 | Change set | 71 |
| 4.5 | Tool support for synchronization models | 73 |
| 4.6 | Summary | 73 |
| 5 | Architectures | 75 |
| 5.1 | Locally to a server | 75 |
| 5.2 | Remote login | 75 |
| 5.3 | Laptop computer to a server | 76 |
| 5.4 | Several sites by Master-Slave connections | 76 |
| 5.5 | Several sites with differing areas of responsibility .. | 76 |
| 5.6 | Several sites with equal servers | 77 |
| 5.7 | Discussion | 77 |
| 5.8 | Summary | 78 |
| 6 | Experiences and advice on key areas | 79 |
| 6.1 | Concurrent development and awareness | 79 |
| 6.2 | Change management | 84 |
| 6.3 | Incremental development | 88 |
| 6.4 | Security | 92 |
| 6.5 | Data storage, repository | 94 |

| | | |
|-----|---|----|
| 6.6 | Integration and delivery | 96 |
| 6.7 | Consistent development environments | 98 |
| | References | 99 |

Paper II - Appendix: Tools and Interviews (in Swedish)

| | | |
|-------|--|------------|
| | | 101 |
| 7 | Forskning & Trender | 101 |
| 7.1 | Forskning & Trender | 101 |
| 7.2 | Konferenser och journaler | 103 |
| 8 | Verktyg | 104 |
| 8.1 | ClearCase | 105 |
| 8.2 | Continuus | 108 |
| 8.3 | CVS | 110 |
| 8.4 | ExcoConf | 113 |
| 8.5 | JavaSafe | 115 |
| 8.6 | PCMS | 116 |
| 8.7 | PVCS | 119 |
| 8.8 | Teamware | 121 |
| 8.9 | TRUEchange | 124 |
| 8.10 | Visual SourceSafe | 127 |
| 9 | Tips vid införande av verktygsstöd | 129 |
| 10 | Intervjuer | 131 |
| 10.1 | Inledning | 131 |
| 10.2 | Frågeställningar | 131 |
| 10.3 | ABB Automation Products | 133 |
| 10.4 | ABB Robotics Products | 139 |
| 10.5 | Combitech Software | 141 |
| 10.6 | Enator | 147 |
| 10.7 | Ericsson Microwave Systems AB | 150 |
| 10.8 | Ericsson Mobile Communications | 156 |
| 10.9 | Ericsson Telecom | 159 |
| 10.10 | Factum Elektronik AB | 162 |
| 10.11 | Kockums Computer Systems | 164 |
| 10.12 | Saab Gripen | 168 |
| 10.13 | Telelogic | 171 |
| | Referenser | 174 |

Introduction

Configuration Management (CM) is a discipline within software engineering with the aim to control and manage projects and to help developers synchronize their work with each other. This is obtained by defining methods and processes to obey, making plans to follow and by using CM-tools that help developers and project leaders with their daily work.

CM has during the past years been considered more and more important due to several reasons. One reason is the influence of the well known SEI Capability Maturity Model (CMM) [SEI95], which have pointed out CM as an important ('key process') area to achieve level 2 on its 5 level scale. Another important reason is the fact that software is getting larger and more complex and needs the support from CM. One example of how the situation gets more complicated is the shorter time-to-market which requires incremental and concurrent development which, in turn, increase the burden on CM. Another example is the trend that developers are getting more dispersed, although still working on the same system.

CM was originally developed under the more or less explicit assumption that the people as well as the files are situated at the same geographical location. This applies to the tools that have been developed as well as to the work processes used. When this assumption no longer is true it creates new demands and tools and processes needs to be re-evaluated. Some aspects, such as file access, have been partly solved by supporting server replication of the files. Other aspects, such as the creation of a general picture and a context, and the communication between developers and groups, have remained manual and without direct tool support. I.e. an astonishingly large part of the requirement of information for synchronization, within, as well as between groups, is covered by informal interaction between developers, during discussions, at review meetings, during coffee breaks, in the corridor, and so on. When people working closely together are geographically disperse we need to consider how these additional aspects may be supported in the work method and in tool support. The closer together people work (despite the geographical distance) and

6 Introduction

the more dependent their various tasks are, the more apparent the lack of support will be.

To geographically distribute the development of a system first sounds as a bad idea. There are, however, reasons to do so and in some cases there is no alternative. Thus, in reality a large number of companies have their developers geographically dispersed, although working on the same system. Some of them have more or less accidentally come to this situation while others have planned it to better utilize resources in their global organization. Irrespective of reason many companies have found that the methods and tools used do not fully support their current situation, and that it will be even worse in the near future. Therefore they now intend to develop such support. We can also see the trend that tool vendors more focus on support for distributed development and teams in general.

1 Research focus

Configuration Management

The area of Software Engineering covers also the sub-discipline Configuration Management [Som97, TF94, BW98, YLM98]. CM is part of the entire development life cycle and is also a very broad area with respect to the means of how to achieve its goals, including everything from methods and processes to file access. One definition of CM is:

CM is the controlled way of leading and managing the development of and changes of combined systems and products, during their entire life cycle.

However, there are several other definitions, all with a different focus. One reason for the differing definitions is that CM has two target groups with rather different needs: management and developers.

From a management perspective, CM directs and controls the development of a product by the identification of the product components and control of their continuous changes. The goal is to document the composition and status of a defined product and its components, as well as to publish this such that the correct working basis is being used and that the right product composition is being made. One example of a definition supporting this discipline is ISO 10 007 [ISO95] meaning that the major goal within CM is “to document and provide full visibility of the product's present configuration and on the status of achievement of its physical and functional requirements”.

From a developer perspective (tool support), CM maintains the products current components, stores their history, offers a stable development environment and coordinates simultaneous changes in the product. The goal is to make a group of developers as efficient as possible in their common work with the product. CM includes both the product (configuration) and the working mode (methods). The definition by Babich [Bab86] stresses the fact that it is often a group of developers that shall together develop and support a system: “Configuration management is the art of

identifying, organizing, and controlling modifications to the software being built by a programming team”.

Distributed development

Our definition of distributed development refers to the situation arising when developers, or groups of developers, who are developing the same software, are geographically dispersed. This includes anything from different parts of the same town to different continents with different time zones. As a result, there is a risk of becoming dependent on a slower and less reliable network, and as a consequence having to copy common files - with all the problems that doing this can cause. In addition, geographical separation results in decreased opportunities for meetings, both formal and informal, e.g. at coffee breaks. This means that the informal interaction between groups becomes reduced, resulting in less knowledge of the overall relationship between the sub-projects. It also means that there is a risk that the connectivity within the group (the team spirit) may be weakened at distribution, this complicates the interaction between the groups when there are problems e.g. with a common interface. Thus the situation makes new demands on how to manage common data, e.g. files, but also how information may be spread between and within groups of developers.

How they interact

The focus of this thesis is how distributed development affects the requirements of CM; what parts of CM are becoming more important (or less) during distributed development compared to local development. Both CM for managers and developers are concerned even though the problem of how to synchronize developers is getting most attention. Both methods of work and the support from current tools are studied.

1.1 Research questions

The physical, technical, deficiencies of distributed development compared to local development are relatively easy to understand and find out. It is harder, however, to realize its consequences. How is the work process affected, are the defined and currently used methods still followed, or do we have to adjust these routines for the new situation. In addition to be aware of the loss of effectiveness and the introduced overhead of distribution, it is also important to understand that it has higher and maybe new requirements, both when it comes to the work process, tool support, and organization (methods, processes, tools, education, etc. needs to be agreed about between sites/organizations).

Examples of questions this study has tried to answer are:

1. How do developers and groups of developers really work today with respect to concurrent and distributed development?
2. How is the work model and use of CM tools affected when the development moves from local to distributed?

8 Introduction

3. What do we need to improve to fully support distributed development?

2 Research methods

The results presented in this thesis are experiences from two empirical studies of how Swedish industry work today in respect of concurrent work and distributed development. As a first step to answer the questions above our goal was to study and understand the work processes of today and note their pros and cons. Next step was to understand if these processes were intended and used because of their good properties or just a result of bad control and/or lack of accurate tool functionality. We also proposed parts of a new work process and identified the tool support needed for this new way of working. Our assumption was that if we have studied the needs of sufficiently many different companies, and found the common needs from these companies, the proposed methods and tool support hopefully could suit many companies. Future research is to implement and evaluate these proposals to get feedback.

The size of the studies are:

- Case-study report: frequent meetings at KCS during a 6 month period consisting of interviews, discussions to understand and critically study their development environment. I also tried the roles of both a CM specialist working with e.g. their ClearCase scripts, and as a developer implementing a small piece of functionality.
- VI-report: almost 10 months of committee meetings and company interviews (each interview consists of a one day interview plus follow-up discussions). 11 interviews resulted in summaries of their own within the report, but also a number of other interviews took place contributing to the experiences presented.

3 Summary of results

There are several results in term of experiences and conclusions in the two reports included in this thesis. In this chapter I will, very briefly, mention some of them.

The main conclusion is that it is always easiest to develop locally and that the distributed development is less effective than local development. Some of the reasons for this are:

- The use of slower and less reliable networks between developers and between the developer and the server. The methods used is often developed for local development and when used in this new situation they may become tedious and time consuming which can tempt the developer to 'cheat' and not follow the defined routines.
- It is, of course, more difficult to arrange formal meetings and there exists no informal meetings (e.g. common coffee breaks). How much these meetings really affect a project is not fully known, but the

experience is that they clearly do influence. A concrete example is the time needed to understand what have happened after a structural code modification made by another developer. The time consumed is normally much longer if the developer is located at another site than if within the same coffee-group.

- Different native languages and cultural differences make misunderstandings more common and the work therefore needs to be done more strict and formal. Sometimes unintelligible deviation from the methods that should be followed or disobey of requests is due to misunderstanding and/or cultural differences.
- Lack of adequate functionality in the used CM-tools. A tool mainly supports one or two models for how concurrent work should be synchronized. A tool that supports a model aimed at local development is not usually optimal for distributed development. Many companies still use their inappropriate tool.

Despite these drawbacks it is, however, sometimes not possible to chose and the trend is that an increasing number of companies acquire a distributed environment. Another conclusion is that it is becoming more common to provide for the prerequisites for distributed development, to be able to make use of the resources within the company in a more efficient way, no matter where they work. Previously, a distributed situation was usually more of a consequence of some other occurrence. e.g. a company fusion.

One of the results in this thesis is the gathering of methods used and practical guidelines for how to compensate the drawbacks mentioned above, and to discuss pros and cons of these compensations. Some of them are listed below:

- Use of telephone and video conferences. These can substitute some formal meetings, especially regular meetings reporting current status. Even if the technique is getting better it is still, however, much harder to discuss and solve problems in a video conference than during a face-to-face meeting.
- A good strategy during all development is to design a good product structure, i.e. to limit the dependency between the developers, especially if they are situated at different locations. The system is divided into modules or components, which are then developed by different groups separately. However, it turns out that despite good structuring, dependencies between the components remain. This becomes clear not least when interfaces need to be modified or when the components are to be integrated.
- To define strict module responsibility, i.e. each file/module has an owner who is responsible for the unit and makes all the modifications to it, solves the problem of concurrent, conflicting, changes. The drawback with this strategy is that it may be too static. If, for example, a change request requires several modules to be modified it is often more efficient to let one developer, responsible for the change request, do all related modifications instead of sending

10 Introduction

request to all involved ‘owners’. This, however, requires some other, more flexible, technique to synchronize concurrent modifications.

- Well defined CM-plan, especially the delivery process. A CM-plan is always necessary, but when distributed it is even more important that everybody knows what should be done, by whom, and when. The plan thus can serve as a ‘passive awareness’ of what is really happening. Moreover, work done at one site is often seen as ‘delivered’ to the other sites, which makes deliveries more common and therefore should be early decided upon and documented.
- Replication of repositories, i.e. to automatically keep replicas on different sites synchronized, allows awareness of what is happening at other sites. It also enables an easier delivery process using merge of branches, i.e. changes made to one branch is integrated (merged) to another branch, instead of using e-mail or ftp.
- Change management system. It is often not a good idea to have copies of change request databases since it is hard to manually keep them synchronized. On the other hand it is practically not possible to have all requests in one central database. One solution is to have many databases organized in ‘layers’ and an automatic process of how a request moves between these layers. Some databases can thus be global, e.g. used by the sales department, and others can be local, e.g. a RCS-repository used by the developers. This also enables the database to be implemented in different systems (Oracle, RCS, etc.), supporting their different needs.

3.1 Shortcomings of approaches to date

The current work processes and the support for distributed development used to date has come to a level of managing co-located groups. I.e. groups of developers at different sites can, without too much problem, develop different parts of a common system. To also fully support the more demanding situation with developers working more closely together being geographically disperse, called ‘distributed groups’ in [Ask99], there is, however, still much to do. Some shortcomings identified are:

- The work processes and tool functionalities are still focused on management and project support, and not at supporting the developers. Examples of identified needs are:
 - support for different types of branches, e.g. ‘light-weight’-branches used for temporary parallel work.
 - fine-grained ‘micro versions’ that can be used by a developer for his/her own daily versioning. These versions can also be used for awareness, i.e. other developers can see them being created, but without being able to use them in their own configurations.
 - process-support of common routines. Not only what to do, but also how to do it. E.g. how integration (merge) to the main branch should be performed (always integrate and test on branch first, before integrating to main).
 - better support for merge, e.g. more viewable merge results

- Many of the solutions to date are based on static product structure and fixed areas of responsibility. In the case 'distributed groups' more flexible solutions are needed, otherwise the case will be more like 'co-located developers' than one group. Identified needs are:
 - Better collaborative awareness. It must be easier to see what other developers are doing and to follow the evolution of the product being developed, not only the individual files.
 - Integrated support for both synchronous and asynchronous collaboration. Within a group of developers both synchronous (e.g. design discussions, code review, etc.) and asynchronous collaboration is needed.
 - Symmetric replication. The replication should be transparent to the users and it should be possible to login at any site without any difference. It should, for example, be possible to create new versions on any branch.
 - Better support for concurrent development at file level, i.e. to facilitate concurrent modifications of the same file and to support merge of these modifications. In this way we do not need the strict responsibility of files/module but can have a more flexible synchronization.

4 Research results

The concrete results from this work has in writing been presented in:

- A Case-Study of Configuration Management with ClearCase in an Industrial Environment. Ulf Asklund and Boris Magnusson. In Proceedings of SCM-7 - International Workshop on Software Configuration Management, R. Conradi (Ed.), Boston, May 1997, LNCS, Springer Verlag.
- Distribuerad utveckling och Configuration Management för programvarusystem. Ulf Asklund. Report published by the Association of Swedish Engineering Industries, 1999. No V040073. ISSN 1493-6444.
- Experiences; Distributed Development and Software Configuration Management. U. Asklund, B. Magnusson, and A. Persson. In Proceedings of SCM-9 - International Symposium on System Configuration Management, J. Estublier (Ed.), Toulouse, France, September 1999. To appear.

The results have also been (will be) presented at several conferences:

- Seventh International Workshop on Software Configuration Management. Boston, USA, May 1997.
- Conference on Configuration Management and distributed development for software systems. Organized by the association of Swedish Engineering Industries. Stockholm, February 1999.

12 Introduction

- SPIN-syd vårkonferens 99. Hur gör man programvara? Organized by SPIN-syd, Software Process Improvement Network. Lund, April 1999.
- Spring Configuration Management Seminar 1999. Organized by the Configuration Management center of competence at Ericsson Business Consulting Sverige AB. Stockholm, May 1999.
- Ninth International Symposium on System Configuration Management. Toulouse, France, September 1999.
- Presentations at Swedish companies as a spin-off effect from the more formal conferences.

4.1 Main contributions

The main contribution in this thesis is the experiences and a state of the art description from two rather big field-studies. There is also a contribution in the work of systematize these experiences, which makes them easier to understand and learn from. Also concrete proposals have been made to meet some of the problems/requirements found during the study.

Some of the contributions can be grouped in relation to the research questions in section 1.1.

1. *How do developers and groups of developers really work today with respect to concurrent and distributed development?*
 - Gathered experiences from more than eleven interviews with companies using distributed development answers this question.
2. *How is the work model and use of CM tools affected when the development moves from local to distributed?*
 - We have classified four different situations (cases) in which distributed development occurs. For each case its CM characteristics is pin-pointed, i.e. in what way the case differ from local development. This classification makes it easier for a company to recognize and analyze its situation and it makes it easier to see how a change (new case) will affect their CM. A real situation is often a combination of these defined cases - and so is the solution.
 - We have named four different client-server architectures and described in what way they relate to the cases above. E.g. can one of the cases ('distributed groups') only be fully supported using one of the architectures ('symmetric replication'). We have also described different synchronization models and how these relate to the cases. Finally we have, for a selection of commercial CM-tools, studied how tools support the architectures and synchronization models, i.e. how they in practice support the cases of distribution, and for a specific company, how different cases are supported by the tool used.
3. *What do we need to improve to fully support distributed development?*
 - Some concrete suggestions to changes of ClearCase are made.

- We have named the case ‘distributed groups’ and identified some new requirements to fully support that situation.

4.2 Summary of papers

This thesis consists of two independent reports summarized below. The second report is originally written in Swedish, but the first part has been translated into English which is the version included here together with the remaining part, the interviews, in Swedish as an appendix.

“A Case-Study of Configuration Management with ClearCase in an Industrial Environment”

A result from a close collaboration for almost six months with Kockums Computer Systems (KCS), where we studied a CM-tool, ClearCase, in practical use. We were particularly interested in the functionality it offers and how it fits the needs of the company - how it has been tailored for use in their environment. The aim of the case-study was to do an objective, independent and critical examination of how a company with well developed configuration management handled their software. We also tried to identify shortcomings and suggest solutions of how they can be overcome in the current framework. We compared the system and the situation with the research prototype we are working with, COOP/Orm, with the goal of understanding how the needs at the company can be supported not to miss important practical aspects of real use. The report is a description of some of the companies work processes and how these are supported by their CM-tool. It is discussed whether the methods used are optimal or rather a result of the current support (or lack of support).

“Distributed Development and Configuration Management”

A report ordered by the association of Swedish Engineering Industries to investigate CM aspects of distributed development; pros and cons, state of the art in Swedish industry and current trends. More than eleven companies were interviewed and the experiences are presented both in terms of practical guidelines and at a higher level of abstraction.

The report gives an overview of CM and distributed development, identifies different cases of distributed development and client-server architectures, and also describes different synchronization models and how they correlate to the requirements identified. Ends with experiences grouped in some CM key areas.

Appendix (in Swedish) Gives a short introduction to current research and trends, a description of a selection of commercial CM-tools, their fundamental model, and how they support distributed development. Also includes summaries from eleven interviews.

4.3 Future research

The results in this thesis is mostly of the type identifying and classifying situations and problems, and how industry manage to date. Left for future research is to find better solutions than used today and to make them used instead.

Still do developers not fully utilize the information stored within the CM-tools. Some information is maybe not of use, but the main reason is that it is to awkward to use in the daily work. In this thesis some solutions to the problem are proposed, but it remains to implement and evaluate them.

The version models used for configurations, i.e. how configurations of versioned files and sub-configurations are managed, still do not provide sufficient overview. In a distributed setting it is even more important that developers, as well as managers, can follow the evolution, not only for an individual file, but for the entire system developed. Much considerable research have been made on version models [CW98, Kat90, Fei91], including our own [MA96, ABHM99], but there are still many open questions to be answered.

Better support for 'distributed groups' is needed, especially to compensate for informal meetings and group awareness. How people work in groups have already been studied within the CSCW community, e.g. by developing and studying 'groupware' and collaborative writing [NCK⁺92, TMG97, BNPM93]. Example of such applications and functionality are synchronous editors, telepointers, group awareness, and WYSIWIS. Also our group in Lund have been working within this area for some years. The main contribution within the CSCW-area is group awareness through 'active diffs' which allows both synchronous and asynchronous collaboration, and a smooth transaction between them [MM93, MA95]. A prototype, COOP/Orm [MAM93], has been developed that implements part of the model. An ongoing project in collaboration with industry will evaluate the model by using the prototype for some small tasks at the company, hopefully with valuable feedback.

Future work is to better integrate the research from the CSCW community with CM-systems to better support collaboration aspects [Grin96].

References

- [Ask99] Ulf Asklund. Distribuerad utveckling och Configuration Management för programvarusystem. Report published by the association of Swedish Engineering Industries, 1999. ISSN 1493-6444.
- [ABHM99] U. Asklund, L. Bendix, H.B. Christensen, and B. Magnusson. The Unified Extensional Versioning Model. In Proceedings of SCM-9 - Ninth International Symposium on System Configuration Management, J. Estublier (Ed.), Toulouse, France, September 1999. To appear.

- [AM97] U. Asklund och B. Magnusson. A Case-Study of Configuration Management with ClearCase in an Industrial Environment. In *Proceedings of SCM7 - International Workshop on Software Configuration Management*, R. Conradi (Ed.), Boston, May 1997, LNCS, Springer Verlag.
- [AMP99] U. Asklund, B. Magnusson, and A. Persson. "Experiences; Distributed Development and Software Configuration Management. In Proceedings of SCM-9 - International Symposium on System Configuration Management, J. Estublier (Ed.), Toulouse, France, September 1999. To appear.
- [Bab86] Wayne A. Babich. Software configuration management: coordination for team productivity. Addison-Wesley. 1986. ISBN 0-201-10161-0.
- [BNPM93] R. Baecker, D. Nastos, I. Posner, and K. Mawby. "The user-centered iterative design of collaborative writing software". In proceedings of InterCHI 93, Amsterdam, The Netherlands, 1993.
- [BW98] Clive Burrows and Ian Wesley. Ovum evaluates: Configuration Management, Ovum Ltd 1998. ISBN 1-898972-24-9
- [CW98] Reidar Conradi and Bernhard Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2):232--282, June 1998.
- [EG89] C.A. Ellis and S.J. Gibbs. "Concurrency control i groupware systems". In proceedings of the ACM SIGMOD Conference on Management of Data. 1989, ACM, New York.
- [Grin96] R. Grinter. "Supporting Articulation Work Using Software Configuration Management Systems". *CSCW: The Journal of Collaborative Computing* 5, 1996.
- [ISO95] Quality management - Guidelines for configuration management. Standardiseringen i Sverige (SIS) SS-EN ISO 10 007.
- [Kat90] Randy H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22(4), December 1990.
- [MA95] Boris Magnusson and Ulf Asklund: Collaborative Editing - Distributed and replication of shared versioned objects. Presented at the Workshop on Mobility and Replication, held with ECOOP 95, Aarhus, August 1995. Available as: LU-CS-TR:96-162, Dept. of Computer Science, Lund, Sweden.
- [MA96] Boris Magnusson and Ulf Asklund. Fine Grained Version Control of Configurations in COOP/Orm. In Sommerville, I. (Ed.) *Proceedings of the 6th International Workshop on Software Configuration Management*, LNCS, Springer Verlag, Berlin. 1996
- [MAM93] Boris Magnusson, Ulf Asklund, and Sten Minör. Fine-Grained Revision Control for Collaborative Software Development. In *Proceedings of ACM SIGSOFT'93 - Symposium on the Foundations of Software Engineering*, Los Angeles, California, 7-10 December 1993.
- [MM93] Sten Minör and Boris Magnusson. A Model for Semi-(a)Synchronous Collaborative Editing. In *Proceedings of ECSCW93, the Third European Conference on Computer Supported Cooperative Work*, Milano, Italy, 1993. Kluwer Academic Publishers.
- [NCK⁺⁹²] C.M. Neuwirth, R. Chanhok, D.S. Kaufer, P. Erion, et.al. "Flexible diff-ing in a collaborative writing system". In proceedings of ACM Conference on Computer Supported Cooperative Work 1992.

16 Introduction

- [SEI95] *The Capability Maturity Model*. Software Engineering Institute, Carnegie Mellon University, Addison Wesley 1995.
- [Som97] Ian Sommerville. Software Engineering. Addison-Wesley. ISBN 0-201-42765-6. 1997
- [TF94] W.F. Tichy and S.I. Feldman (Eds.). Configuration Management. ISBN 0-471-94245-6. John Wiley & Sons. 1994.
- [TMGS97] S.G. Tammaro, J.N. Mosier, N.C. Goodwin, and G. Spitz. “Collaborative writing is hard to support: A field study of collaborative writing”. CSCW: The journal of collaborative computing 6, 1997.
- [YLM98] André van der Hoek (Ed.). Configuration Management Yellow Pages. http://www.cs.colorado.edu/~andre/configuration_management.html

A Case-Study of Configuration Management with ClearCase in an Industrial Environment

Ulf Asklund and Boris Magnusson

Published in Proceedings of SCM7, International Workshop on Software Configuration Management, R. Conradi (Ed.), Boston, May 1997, LNCS, Springer Verlag

Abstract

This paper reports from a case study where the configuration management system ClearCase is used in a large scale industrial application. The focus of the study is on the functionality offered and how it matches the needs in this particular situation. The paper reports on situations where ClearCase has turned out to be cumbersome to use or is lacking functionality. Improvements are outlined for how the problems can be solved or the situations better supported. The suggested improvements are influenced by experience with the COOP/Orm research prototype and some of the improvements are illustrated with functionality available in this integrated environment.

1 Introduction

Configuration management is a well-known problem in industry since most products evolve over time. There is a need to keep track of which components are included in a specific version of a product. The car industry is a typical example where each car model is often revised every year and versions of spare parts for older versions of a model needs to be identified during a long time period.

The situation in the software industry is at first sight similar, but it turns out to be much harder to cope with. Explanations for the increased difficulties can be sought in aspects such as a much faster change of versions (including internal versions), less visibility of incompatibility (compared to mechanical parts), very complex systems, and less standardization of parts and sub-systems. In academia and in parts of the software industry these problems have been identified and systems for management of software components have been developed. A first generation of tools, such as SCCS [Roe75], and RCS [Tic85], were based on version control of single files. These systems give limited support, but are simple to introduce and to use. A second generation of tools, which take a broader view on support in the software development process, such as Continus/CM [Con98, Cla95], ClearCase [Clear98, Cla95], and Teamware [SMS], are now finding their way into industry. While giving more support they also have a larger influence on the developers' work process and are thus harder to introduce. The acceptance of such tools is a comparably slow process and seems to have taken place mainly in some kernel software industry. Many industries are now, however, facing the situation that software components are part of what they used to think of as mechanical or electronic products. As the software components grow, the traditional methods for version control and configuration management may prove inadequate and there will be a need for more sophisticated software configuration management tools in a wider community. At the same time there are research activities to produce even more sophisticated environments and tools. There is thus an interesting question of to what extent the existing systems meet the needs for configuration management support, in the pure software industry as well as in the traditional industry.

Introducing configuration management tools in an existing organization is not always easy. Such tools inflict some overhead on the developers in their everyday short-term work, which they may not recognize as motivated. The benefits seem to be more visible for managers, and in the long run. It therefore often takes an explicit management decision to introduce configuration management in an existing organization. The needed understanding of CM problems, principles, and systems is more likely to occur in large-scale software companies, but the need for CM is equally important in smaller companies. In introducing configuration management tools it is important that the tools are appreciated as a help rather than as an additional burden for the developers. They must thus be easy to use and understandable in terms of the development process used at the company. In a large company there might be resources to create a specialized support group for configuration management, but in smaller com-

panies these tasks must be handled by the developers themselves. There is thus also a demand that the tools and systems must be easy to manage. Tasks like creating new development lines, branches, merging and integration of development lines, finding out what development lines exists, etc. must be possible to do without too much training and work.

In this paper we report from the use of a second generation system, ClearCase, in a software company. We have been particularly interested in the functionality it offers and how it fits the needs of the company. The aim of this case study was to do an objective, independent and critical examination of how a company with well developed configuration management handled their software. We have tried to identify shortcomings and suggest solutions of how they can be overcome in the current framework. We have also compared the system and the situation with the research prototype we are working with, COOP/Orm. The goal here is to understand how the needs at the company can be supported in order not to miss important aspects of real use. We outline how the model used in the COOP/Orm system can go further in the support offered, in particular in the areas of providing group awareness, fine grained version control and supporting synchronous interaction.

In section 2 we give some background on the company, what they do and the environment they work in, a short introduction to ClearCase and to the COOP/Orm prototype. In section 3 we describe in more detail how ClearCase is used today, including local adaptation. In section 4 we identify and describe particular problems with the use of ClearCase. In section 5 we outline how the same situation could be handled in an environment with the facilities available in COOP/Orm. In section 6 we outline solutions of how the problems could be solved or reduced in the ClearCase framework. Section 7 discusses directions for future work, and section 8 concludes the paper.

2 Background

2.1 Introduction to ClearCase

ClearCase by Atria is a version-control and configuration-management system, designed for development teams working in Unix on a local area network. ClearCase MultiSite extends the ClearCase features to also support geographically distributed development teams. ClearCase stores all data under its control in versioned-object bases (VOBs). A VOB is an implementation of the NFS [NFS] protocol and access can thus be physically distributed in a local area network. VOBs also inherit from NFS the restrictions that they can not span physical discs. Each file and directory visible through a NFS-mounted VOB appears as a Unix file or directory which gives a high degree of transparency to standard tools. A VOB can be replicated to remote sites, but there remains a notion of ownership with one of the sites.

Versioning is always in the context of a particular *branch-type*. A branch-type is identified with a name and is relevant for all files in a

VOB. Initially there is only one branch-type, ‘main’, but at any point in the development a new branch-type can be created. There is no assumption of any relation between branch-types in ClearCase. A file can exist in a sequence of versions on a branch-type (often called a ‘*branch*’ of the file). A new version of a file in a particular branch-type is created by ‘*checking out*’ the file. A file can be checked out, exist, on several different branch-types at the same time. Synchronization within a branch-type is by locking, preventing developers working on the same branch-type to simultaneously change the same file. Versions of a file on different branch-types can, however, be checked-out in parallel.

All users access versioned source data in the VOB through a view, and a virtual *workspace*. A View provides a selection mechanism and a workspace is a storage area (directory) in which developers can perform tasks in isolation from other development. Each view has an associated *configuration specification* which lists rules for selecting a version of each needed file. Many rules name a branch-type on which to look for a version of a needed file. The rules are evaluated in order until a rule matching an existing version of the file is found. They can thus be used to specify an order among the branch-types to use in the particular view. Each rule can be dynamic, allowing users to see, for example, the latest version of a file on a branch-type, or it can be fixed to allow a developer to work with a fixed version of a file regardless if later versions of the file exists on that branch-type, or it can simply name a particular version of a file or a ‘label’ (as of below).

A development project typically use a particular branch-type and a view. Files that have been changed in the project have versions on that branch-type. These versions are often thought of as a branch (variant) of the file. The view specifies a dynamic rule to select files that have versions on the branch-type, perhaps in several steps, and at the end some fixed rule, which is used as default to get a stable version (such as the latest release) of files that have not been changed in the current development project. Views that only contain fixed rules will always return the same version of all files and can thus be used to represent a version of a configuration, a baseline.

Merging of branches is done using the ClearCase merge tool. It identifies the differences between the branch-types to merge in terms of files that have been updated on the branch-types. For each of these files it identifies a ‘common ancestor’ version of the versions used on the branch-types. Files that are changed in only one branch are included in the updated version. Files that have been changed in more than one branch, but where the changes do not effect the same original source line, can be automatically merged. For files with changes on the same source line, the developer is prompted to choose which changes to accept in order to resolve the conflict.

Views are used to select a configuration of a system, but since this selection is done on demand it gives different results as new versions of files are created. In order to make it possible to come back to a particular set of versions of files there is a need to identify versions of configurations. This is done through *labeling* - all the versions of files in such a set are marked with the same label (such as “Release.2”). A view can use labels to

select versions of files to recreate the configuration. Although views and thus branch-types can be used to name a labeled configuration, there is afterwards no relation between the label and the branch-type or indeed between the labeled configurations. Branch-types are thus only a kind of common naming for variants of files, but do not support organizing versions of configurations.

For replicated VOBs, branch-types come with a protection mechanism which restrict creation of versions of files on a particular branch-type to one site. Versions of files on the built in branch-type, 'main', can only be created on the site that acts as the master of the VOB.

A triggering mechanism can be used to extend the functionality of ClearCase by scripts executed at situations such as check-out. This facility can be used for all kinds of extensions such as logging, restricted access, work process, etc.

Small example

We will with a small example further illustrate the functionality of ClearCase. The example also describes a common work process of how branch-types are used.

Consider a small system which consists of ten files of source code, named foo1.c to foo10.c, all stored in the same directory. In the initial situation the program has been developed within ClearCase, but without using any branch-types and that the latest version of all files are labeled with the label 'Beta1.0'. I.e. all files, including the directory, have all been versioned on 'main', where they may have reached different version numbers. Thus, each file has a version tree as depicted in Figure 1a, where x is the latest version for that file. The next step in the development of this program is to implement new functionality but also to correct bugs reported from the beta testers. These two tasks should, by the two developers Ulf and Boris, be accomplished in parallel and their individual results should then later be merged and labeled forming the first release.

For the purpose two branch-types, proj_A and bug_fix, are created. Ulf, who is responsible for the new development, has the following configuration specification in his view:

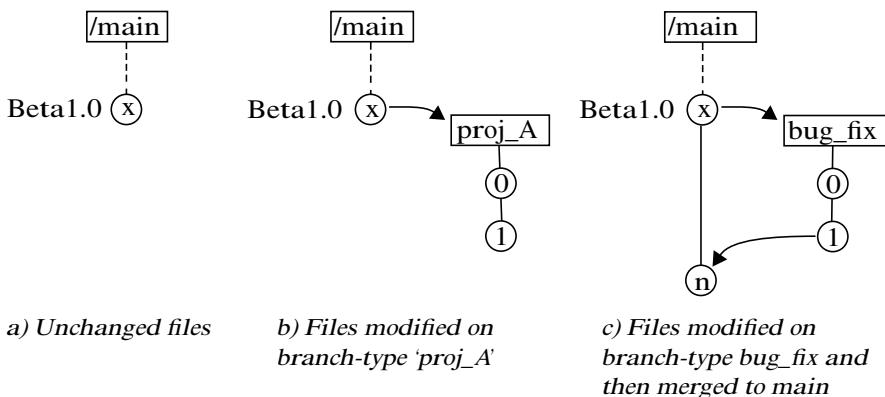


Figure 1 Version trees for individual files

```
element * CHECKEDOUT
element * .../proj_A/LATEST
element * /main/Beta1.0 -mkbranch proj_A
```

When a file is needed, the rules are evaluated top-down until a matching version of the file is found. A rule, i.e. a line in the description, consists of four parts: type of item evaluated by the rule, restriction on files to be evaluated due to their place in the structure, the rule, and (optional) the branch that should be created when a file chosen by this rule is checked-out. The rule itself can contain either a specific version, a label, or one of the predefined functions: 'CHECKEDOUT' or 'LATEST'. The first rule will select a checked out version of a file. In case there is no such version, the second rule will select the latest update of the file on the branch-type 'proj_A'. Finally, for files that have no version matching either of these, the version of the file labeled 'Beta1.0' will be selected.

Within the view Ulf implements the new functionality in isolation from Boris' changes. To accomplish his task foo1.c and foo2.c are checked-out, modified and checked-in. Thus the two files have a version tree as depicted in Figure 1b. The files and the directory not yet modified (if not Boris has created new versions of them) have the same version tree (Figure 1a). I.e. a branch is created on demand for each file. We call the creation of a branch, i.e. when, for a specific file, the first version on the branch is created, step A in the work process.

In parallel with Ulf's work Boris check-out, modify and check-in the files foo2.c and foo3.c to accomplish his task. When the bugs are fixed and tested the bug_fix branch-type is merged to main. Merges are also visible in the version tree, depicted by the file foo3.c version tree in Figure 1c.

Ulf just finished his task and is now ready to merge his changes to main. However, to avoid incorrect files on the main branch-type the work process convention is that integration tests must be made on the branch-type before merging to main. This means that before a merge to main is performed, the program should first be merged to the development branch-type and tested there, step B. Possible errors due to merge conflicts can in this way be corrected on the branch-type in isolation from all other projects.

Before merging the configuration specification must be changed to select the latest version on main instead of the labeled one. I.e the specification must be changed to:

```
element * CHECKEDOUT
element * .../proj_A/LATEST
element * /main/LATEST -mkbranch proj_A
```

When the program has been tested and checked-in it can now be merged to main, step C. The specification is again changed, now to:

```
element * CHECKEDOUT
element * /main/LATEST
```

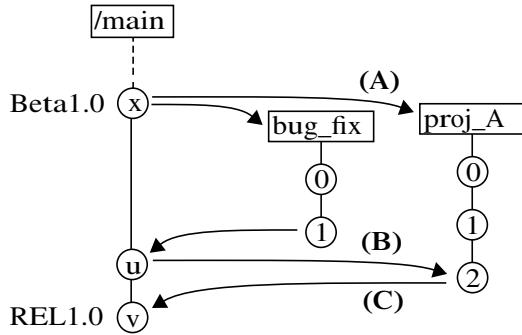


Figure 2 *foo2.c after merge of two sets of parallel changes to main. The changes in the second set are first tested locally (B) before merged into main (C).*

and the merge is made. If no further versions have been created at the main branch-type since step B, this merge is trivial, otherwise step B should be repeated. After the merge to main (C) the program is again tested, files are checked-in, and the selected versions are labeled. By labeling the versions a new version of the entire system is created and can later easily be recreated.

Figure 2 depicts the version tree of the file foo2.c. Both Boris and Ulf modified this file and therefore versions of it exists both on branch-type bug_fix and proj_A. The work process is also depicted with its three steps: (A) the branch-type is created, (B) updating the branch-type from main, and (C) finally merging the branch-type to main. For this file all three steps are visible in the version tree. However, if, for a file, no new version had been checked-in between step A and step B, step B had not been visible in the version tree of that file.

Further details about ClearCase can be found on Atria's www-site [Clear98]. ClearCase is also one of the configuration management tools evaluated by Ovum [BW98].

2.2 Kockums Computer Systems

Kockums Computer Systems, KCS, markets design and production information systems specifically created for the shipbuilding industry. Systems from KCS are currently in service at more than 260 sites in 38 countries in Asia, Australia, Europe, North and South America. KCS' main office is located in Malmö, Sweden, with subsidiaries in Germany, Japan, South Korea, United Kingdom, and the USA.

Their product, TRIBON, is a CAD/CAM/CIM system and is in fact 17 different products combined to one complete system. TRIBON is ported to several hardware platforms including VAX, ALPHA, HP, IBM, Sun, and PC (partly). These platforms in turn have several operating systems in several versions, e.g. HP-UX 9, HP-UX 10, AIX 3, and AIX 4. The programming languages used are among others PL/1, Fortran, C, C++, UIL, and assembler. They, of course, also exists in several versions. These aspects in combination with a geographically distributed development with approximately 60 developers in three different countries makes it a

complex development environment interesting to study. To handle this complex development situation KCS has had a long history of emphasis on configuration management resulting in a well developed CM-model, implemented in ClearCase and ClearCase MultiSite.

2.3 The COOP/Orm environment overview

COOP/Orm is an environment supporting collaborative writing of hierarchically structured documents. It is based on previous work in the Mjølner-project [KLMM93], concerning object-oriented software development. The environment developed in Lund, Mjølner Orm [MHM⁺90], supports collaborative software development to a limited extent through its configuration management [Gus90]. The current project, the COOP/Orm environment, is a research prototype aimed at providing support for distributed teams of developers. It has been influenced by advances in the CSCW area, but its main target area is software development.

COOP/Orm is designed for use in a distributed environment and is built with a multiple-client, multiple-server architecture. Some of the aspects of distribution and replication of data was reported at PDCS-96 [MG96] and in [MA95]. As a result of the design, users of COOP/Orm can have the same support and interaction whether local or distributed although the speed might depend on the quality of the long distance network.

COOP/Orm is also designed to support versioned configurations of documents (such as programs). Links from one document to another is always denoting a particular version of that document (which might be a new configuration). As a result, selecting a version of a document determines particular versions of all (recursively) referenced documents. The model thus support a baseline approach, but the user interface makes it very simple to switch to use another version of a configuration as described at last years SCM [MA96]. Versions of configurations can be interactively compared and differences presented (added/deleted dependencies, change of targeted version).

COOP/Orm supports hierarchical composition of documents in an integrated storage model as described in [MMA93]. This storage model is compact since unchanged parts (or sub-trees) can be shared among many versions. The model support very fast identification of changes in terms of structural changes and changes to the content of a node [Ask94]. The hierarchy can be used to support structured software as e.g. classes and methods (or modules and procedures), but fits equally well for documents with chapters, sections, paragraphs etc. The hierarchical model support automatic merge of variants. Changes such as addition, deletion or change of a part in only one of the variants are handled according to default rules. Parts that have been changed in both variants has to be dealt with by the editor for that particular kind of node (also here default rules are supported).

COOP/Orm maintains a version graph for each document which in the user interface can be used to (very fast) switch between viewing different versions of a document and to compare (also distant) versions. The ver-

sion graph is shared among all users and is updated when any user creates a new version of the document.

For the full benefit of the COOP/Orm model editors for the leaf nodes (such as a text editor) must be version aware in order to support the fine-grained and interactive version control mechanism. As an alternative documents can be written to normal files, edited with some standard editor, and then read back in, at which time the changes will be visible to others.

In contrast to other version-control/configuration-management systems, COOP/Orm offers a high level of collaborative awareness. These aspects are described in more detail in [MM93] and [MMA97]. In the present setting we like to mention a few aspects.

The version graph makes it possible for every user to get an overview of how his version of a document relates to other versions, and to see as new revisions and variants are created by other users. He can also compare his version with other versions, including these under construction as an extreme case. The model also supports synchronous interaction, but that is not further elaborated here.

The support for versioned configurations makes the user aware of when new versions become available. Switching to use another version of a configuration is simple and explicitly visible, and it is as simple to switch back. Selecting a version of a configuration implies selecting versions of all files included in the configuration (possibly very many), and at the same time ensures that a meaningful collection of these files is used ('meaningful' in the sense of an intended combination of versions).

3 CM overview at KCS

3.1 The CM history in the company

Like almost every company the developers at KCS first built their own configuration management system. Within the system, called MMS, the product structure were made. By use of name-conventions, files with functional relationship were structured into a family stored in the same directory. The system had functionality to, by the file name, find the directory containing a specific file and present the header. It was also possible to, very quickly, find dependencies between files (a useful functionality not yet fully implemented in the new system). Until recently all KCS customers had VMS platforms, but Unix use grew and the demands on porting TRIBON to other platforms increased. Unfortunately MMS only run on VMS and KCS therefore started to look at commercial alternatives, of which ClearCase were chosen.

1994 the, well planned, conversion from MMS to ClearCase started. For about 6 months ClearCase was used only as a secondary system and development was still made using MMS. In this way the developers could learn the new system gradually and not necessarily all developers at the same time. Even when ClearCase had become the primary system, MMS was used as backup. Platforms were ported to ClearCase one at the time.

Because the conversion was made gradually and the old product structure remained there was not much resistance from the developers. In fact, the structure and the name conventions are still used.

3.2 How ClearCase is used today

All source code is under the control of ClearCase, i.e. both code that make up the product, and all internal routines and scripts. The source is organized in total 12 VOBs, due to the technical restriction that a VOB can not span physical discs. So far there is no logical or structural background to on which VOB which information is placed. The same file type is used for all code, independent of platform and programming language. Such information are instead obtained by structured comments in the file header. A CM support group has been formed in order to maintain specialized competence and off load the developers from CM tasks. The CM-group works with trouble-shooting and there are particular actions that usual developers are not allowed to perform.

Branch-types are used at KCS for three basic purposes: (1) each development project has its own branch-type, (2) maintenance is done on product branch-types, (3) correction of serious urgent errors are made on a special bug-fix branch-type. Development projects are typically relatively long-lived activities involving several developers. New branch-types are created by the CM-group only, which also help in creating views.

A build of a system is managed through a preprocessor that with a given view extracts the code for a given platform and activates the corresponding compilers etc. Management of variants for different platforms are thus done outside the control of ClearCase, but it enables storage of all code in the same place, i.e. on the same VOB. E.g. are the same discs mounted both on VMS and Unix systems. This enables even VMS code to be under control of ClearCase (which is not ported to VMS). ClearCase is however not used to handle documentation. The reason for this is that ClearCase only fully supports ASCII files and not binaries like word processor files, since the used delta techniques and merge tools do only work on ASCII.

The CM-group has built a command interface on top of ClearCase in order to make it easier to use for the developers and to give additional support. One example of a command is 'cmmkelem' which creates a new file. Before the new file is created a check is made to check that a file with that name does not already exist in some other branch-type. ClearCase would allow this, but it would create problems during merge. The command also supports KCS conventions for naming and organization in directories. Emacs has also been extended to support CM functions.

4 Requirements and shortcomings of ClearCase

The analysis is made with focus on some key areas. In this section each of these areas is stated, and the way that particular problems are handled at

KCS is described and reflected upon. A more elaborated suggestion to changes (improvements) is made in section 6.

4.1 The version selection problem for developers

The problem concerning the selection of files and directories to be included in the workspace and to determine which version of each of these that should be chosen is often called the *selection problem*. The problem increase with the number of files and directories. One core functionality of ClearCase is the configuration specification in a view. A specification filters out the correct version of every file and directory to the developer. The idea is that once the view is set everything should almost be like 'normal', i.e. like working alone. 'Set and forget' is the slogan used in ClearCase manuals. The specification actually serves two roles: define the selection rules, and define the actions to be taken when a file is checked-out. Thus, a configuration specification is very powerful. It seeks through an amount of files and directories a developer hardly can get an overview of and comes up with the result. One danger, however, with this kind of global search approach is that as long as the specification is correct nobody care about how it works, but when then an error occur (which, sooner or later, always happens) it can be hard to solve, or even to notice.

The most frequently encountered CM-related problem at KCS is that the view specification is wrong. Even if the syntax is easy to understand the specification gets complex and it can be hard to see the consequences of a written specification. I.e. it is hard both to verify that a given specification is correct, but also to notice when there is an error. A misspelling, for example, can lead to that completely wrong versions are seen and used for a long time without notice. Even if errors are not so frequent at KCS, they have resulted in a suggestion that only the CM-group should create view-specifications, instead of the developers themselves as today. This will, of course, reduce the CM-knowledge of the developers even more. An observation is that a semantic check of the specifications, finding errors like misspelled branch-type names should reduce the number of errors. However, even with such an improvement the more serious problems of missed, or specifying the wrong, branch-type still remains. A better overview of how the branch-types relate to each other might help, but this is not supported by ClearCase at all.

More support for structuring could make it possible for the developers to get an overview so that they, by themselves, can do the selection or, at least, verify that the selection automatically made is correct. ClearCase supports structure through directories. Versioning of directories is only related to which files are included in a directory, not which version of these files. For decreasing the complexity in the version selection problem ClearCase directories thus offer no help.

Summary – needed support:

- Semantic check of view-specifications
- Support for relations between branch-types
- Support for versioned sub-systems (i.e. directories)

4.2 Support for short term sub-project

At KCS development is divided into several projects. Each project has its own branch-type which makes it possible for developers in different projects to work in parallel. In addition there is a branch-type dedicated to bug fixes. Branch-types are created by the CM-group, but any developer can create a version of a file on a branch-type. All developers working on the same project are working on the same branch-type which makes it important to know how their changes are shared. There are two fundamental strategies: *optimistic* where changes are shared as soon as they take place, and *pessimistic* where changes are seen by others only after their own consent. In ClearCase the strategy is controlled by the views as we will see below.

There are two basic problems elaborated upon in this section: how and by whom branch-types are created, and how to support optimistic and pessimistic propagation of changes within projects.

Strategies for branch type creation

At KCS creating a branch-type is viewed as creating a new project, which is nothing a developer does, but is restricted to be made only by the CM-group (having consulted the product managers). The CM-group thus maintains the knowledge of which branch-types that exist and for what purposes. The drawback is the extra overhead for the CM-group that runs the risk of becoming a bottleneck, and that the developers perceive creating a new branch-type as a heavy operation. Even with these restrictions there are many branch-types and a risk that merge problems occur due to related changes made in different branch-types. Synchronization within a branch-type is obtained by locking – it is thus impossible for developers working on the same project to work in parallel on the same file (although this is possible between project with their own branch-types).

There is a need for developers in the same project to work independently for shorter times. This could in principle be supported by using branch-types, but because of the management overhead on creation of branch-types, motivated by its main use, this would be too heavy-weight.

Baseline vs. generic configurations

A *baseline* means (following the terminology of Tichy [Tic88]) a configuration that specifies the versions of all included files and directories. A *generic configuration* on the other hand is dynamic and is set up using rules rather than fixed versions. Both these strategies can be supported in ClearCase views. Generic configurations are supported with the built in rules 'CHECKEDOUT' (matching the versions in the workspace) and 'LATEST' (matching the latest version of a file in a branch-type). Baselines are supported by using fixed versions and labels as rules.

Returning to the first configuration specification in the example in section 2.1, we can see how a view can be written to specify an optimistic change propagation strategy.

For developers working in the same project there seems to be at least three levels between using a baseline and a generic configuration for all files.

1) The most generic configuration is when the developers use the *same view*. This results in all having exactly the same workspace, i.e. even the same checked-out files are used although only one developer can actually change a checked-out file.

2) 'Normal' generic configuration is obtained by using *private views*, but with the rule to select the latest version in the project branch-type (this case is illustrated in the example view in 2.1). In this case all checked-out files are private and they get visible to (and are used by) other developers in the project when checked-in.

3) Finally, it is possible to work in a baseline configuration, which is obtained by having each developer work alone on a *private branch* (or have a private view with rules selecting specific versions). In this case changes made by other developers will not be incorporated without the user changing his view.

The positive aspect of working in a baseline configuration is that the developer has a stable environment, all differences in behavior of the developed system are related to changes he/she makes. The disadvantage of using baselines during development can be that changes are integrated late and get less tested together with the other developers changes.

Generic configurations, on the other hand, support early testing. Changes from other developers are seen when checked-in and are included during the next 'make'. The disadvantage is that this happens without any knowledge by the developer using the new file. A bad change by one developer may hinder the others on the project until the problem is fixed.

The second strategy put priority on early testing, but provides some protection between developers and is the model most frequently used at KCS. The drawback is that the individual developer can not get any version control between him checking out and checking in a file. Such fine-grained 'micro versions' would often be useful when rolling back a change that did not work. Support in this situation would also have the benefit of offering direct support for the individual developer. The only way to do this in ClearCase would be to introduce private branch-types for each developer and use a policy to do frequent merges to the project branch-type. For reasons mentioned above the management overhead when creating a new branch-type is too high to make this viable.

Summary – needed support:

- 'light-weight' branch-types
- fine-grained 'micro versions'

4.3 Merging and integration

According to the Ovum report [RBI95] ClearCase 'sets a new standard for industry' when it comes to the interface to the merge-conflict resolution. ClearCase thus offers a well developed support for merge. The command used is called 'findmerge' which means that when programs (or entire vobs) are merged ClearCase first *finds* which directories and files that have to be merged and can then, with the developers permission, really do

the *merge* according to its default rules. After the ‘find’-phase ClearCase returns the files and directories that needs to be merged. The result is, however, just a list of names which makes it hard to get an overview. The form of the result seems more suited to be parsed by ClearCase again, launching the merge of the required files, rather than by a developer searching for inconsistencies. Considered that merge and integration often are made just before the release dead-line, a not very readable merge result often results in that the merge is done more or less in blindness, and as much automatic as possible. However, when conflicts are detected by the system and the developer is prompted to resolve it, the graphical merge tool is easy to use. Despite that the merge is line based and not so fine-grained, it is easy to understand and gives required support.

In section 2.1 a common work process, integrating modifications made at a project branch-type to the main branch-type, was illustrated by three steps. The process described is used at KCS, in the way that the CM-group has written conventions followed by the developers. Notable is that the configuration specification must be changed for every step in the process. This means that the specification, which is the single feature most prone to errors, must exist in three versions. A more directed support for this kind of common process should, most certain, reduce the number of errors made by the developers.

One should note that the process is iterative. After the tests have been made on the project branch-type and the next step should be to merge to ‘main’ the developer must make sure that no newer versions have been checked-in to ‘main’ since he did step (B). This is done by doing the find-phase of step (B) again, thus iterating the step (B) and test phases until the latest versions on ‘main’ have been tested and then quickly check in. Even with this iterative process we can not be totally sure that no files have been checked-in before we run step (C). If this happens, which is noticed by carefully reading the merge result, the convention on KCS is to revoke step (B) with an uncheck-out and do step (B) over again. Here a more active awareness of version checked-in had been of great help, see the awareness discussion in section 4.5.

Again, all steps can be made in ClearCase. However, when conventions must be used to force the developers to a certain behavior, we see that as a lack of support.

Summary – needed support:

- Support for integration of projects with a ‘main’ development line.
- More viewable merge results.

4.4 Support for delivery and consistent configurations

In ClearCase (like in e.g. RCS, SCCS, and CVS [Ced93, Wat]) keeping track of configurations are done by labels. I.e. for each file and directory a label, which is a special type of attribute, is attached to the version included in the configuration. It is then easy to ‘rebuild’ this configuration by just selecting the version of each file which is marked with the label. It

might come as a surprise that in a configuration management system there is no support to record relations between such labeled configurations. For files each version is related to other versions of the file. A label, however, is just a text string (such as 'REL_2.0') and other means have to be used to keep track of existing configurations (i.e. Labels), why they are created and their relations.

KCS uses labels to label the version of all files included in a delivery, and also to mark configurations of systems and subsystems that have reached a certain level of maturity. In this way developers working on a subsystem easier can select correct versions of other subsystems without having detailed information about their development status.

Labeling files can be seen as taking a snapshot of the configuration. Between the snapshots, files are checked-in, both to branch-types, but also to main, changing the configuration continuously. It is therefore important to remember to label a configuration that later should be accessible. Normally this is done after a merge that has been tested, but before other changes has been made. It can otherwise be very difficult (although in principle possible) to 'find back' to the desired configuration.

When using labels the aspect of disc space consumption must be considered since labeling implies storing information about all files included in the configuration. With a product like TRIBON, including about 30,000 files, a labeling of the entire system (the main branch-type) will cost about 500 Kilobyte and take about 90 minutes to perform. Even if labeling is an easy operation to invoke the technique can not be used freely by all developers, but must again at KCS be controlled by the CM-group.

There is a need for individual developer to create stable versions of a system during development, for debugging, evaluating the impact of changes compared to earlier versions, backing out of mistakes involving several files etc. With labels these stepping stones must be pre-planned and created relatively often to be meaningful. Unfortunately the labeling technique is too heavy for routine use by individual developers. Again we notice that this is a situation where developers could directly benefit from the use of a CM system, but where well motivated restrictions makes it not possible.

Summary – needed support:

- a more powerful support for configurations than 'labels'

4.5 Support for group awareness

An important aspect of a system for people working together is its support for 'collaborative awareness', i.e. how and to what extent actions performed by others can be noticed by a user. In a CM system there are different demands from managers and developers and there are trade-offs between supporting isolation as opposed to awareness. The needs are also different in a setting where all users are working relatively close (say in the same building) or geographically distributed (as supported by ClearCase MultiSite). In the former case other mechanisms, such as informal

meetings etc. can replace functionality in the CM system while in the latter case demands are higher.

Developer support needs

Development projects at KCS are often organized so the developer(s) can work in relative isolation from developers on other projects. Even so, there are situations where a developer needs to be aware of what other developers are doing. A frequent problem is to see if a file is already checked-out. This is supported in ClearCase with a tool creating graphs for individual files showing in which branch-types versions of the file has been created. Another tool creates a listing of all files currently checked-out. These tools thus contribute to awareness through giving an overview picture.

Using an optimistic update model, one developer checking in (or merging in) files on one branch-type might as a result change the version used by a developer working on another branch-type. In case there are incompatibilities, problems noted by surprise by the second developer, there is a need for support in making him aware of the change, at least a simple way for him to identify that a change has occurred. The ClearCase overview of versions of a file can be used to *verify* that a candidate file has indeed been changed, but there is no support for finding such candidates.

When a new branch-type is created other developers might need to update their views in order to include the new branch-type, although they are not primarily involved with it. Some kind of awareness of this kind of changes are needed. There is no support for this aspect in ClearCase and at KCS the conclusion is that also views might have to be managed by the central CM-group.

The experience at KCS is, however, that not much of the existing functionality in ClearCase supporting ‘awareness’ in the broad sense is in fact used, at least not by the regular developer. The exact reason for this is unclear, but it seems to be a combination of lack of match with the required support and that the provided commands are too awkward to use or have too long execution time. Instead most awareness is achieved by just ‘knowing’ what other developers are doing, i.e. through ‘social protocol’ rather than by system support.

Manager support needs

For the CM-group, managing the use of ClearCase, there is a need for overview of the system, in particular which branch-types that exists and how they are related to each other and to the views and labels. These aspects are poorly supported by ClearCase and at KCS some additional support has been developed by the CM-group. A tool generates html-documents which are stored on a local web-server. In these documents all currently existing views, branch types, and label types can easily be viewed by all developers. Viewing the current views, for example, also shows the corresponding configuration specification. This is to great help when creating new views and reduce the number of errors occurring in the specifications. The generation takes some time, though, and is therefore not made after every change which can result in some confusion when the list is not up to date.

Summary – needed support:

- collaborative awareness that a new version of a file has been created
- collaborative awareness for created branch-types
- relations between branch-types

5 Relevant functionality in the COOP/Orm environment

In this section we will outline how the ideas used in COOP/Orm can solve the problems identified at KCS. Since the problems are expressed in terms of functionality in ClearCase the mapping is not always direct to COOP/Orm functionality. In some cases the problems simply do not exist since they arise from using ClearCase itself. The presentation will follow the organization in section 4. A general observation is that while ClearCase seems more geared towards the needs of managers, COOP/Orm seems to support the needs of the developers as well. This is an important aspect when introducing a CM system, getting it accepted, understood and used.

Version selection problems (4.1) and need for consistent configurations (4.4).

The versioned configuration approach used in COOP/Orm seems to solve many of the problems identified in ClearCase. The explicit choice of versions replaces the need for ‘view’-specifications and thus the problems they introduce. The need for repeated and error-prone changes to ‘view’ specifications is thus replaced with explicit selection in the version-graph. The explicit version graph in COOP/Orm makes the relations among revisions and variants explicit and solves the problem of relations among ‘branch-types’ in ClearCase. The need for versioned sub-systems is directly answered by the versioned configurations in COOP/Orm.

Support for short-term project (4.2).

The reasons not to allow developers to create new ‘branch-types’ in ClearCase are eliminated (or at least greatly reduced) by the explicit version-graph in COOP/Orm. It enables everyone to get an overview of the development history of a file or configuration. It should thus be less problematic to allow users to create short-lived branches. Furthermore, the fine-grained version control mechanism directly supports the micro-versions for the benefit of developers.

Merging and integration (4.3)

The work process for merge followed at KCS, (see Figure 2), can be supported by choosing the order in which merge is performed in COOP/Orm. In addition the version graph will document that the process was followed. Support for the work process could also be improved, so for example the tool would insist on that the agreed process was followed. This is not an aspect in comparison with ClearCase, but is relevant when comparing with other systems such as Teamware [Team] and Continuus [Cont] that to some extent support work processes. The merge support in

COOP/Orm is designed to give a good overview of the result and to let the user interactively explore different possibilities before settling on a particular choice. These mechanisms are currently extended to better support selection of consistent sets of related changes from the variants merged.

Support for group awareness (4.5).

COOP/Orm directly supports the demands for collaboration awareness identified at KCS, when creating a new version of a file as well as when creating a new variant. COOP/Orm go beyond these demands with support for much finer grain of collaborative awareness. The need for this kind of interaction has not yet arisen at KCS, probably due to that project teams are typically put together with local members.

6 Suggested improvements in ClearCase

Some of the problematic situations identified in the previous section are such that there is in principle some functionality in ClearCase that seems to support the situation. However, it turns out in a number of cases that the functionality of ClearCase after all can not be used for the purpose due to some conflicting goal at KCS. In the following we will suggest some technical improvements of ClearCase which we believe is in the line of its overall design and will solve the problems.

6.1 Semantic checks for views

When evaluating a view it should be simple to report to the user any branch-type or label name that does not match an existing name. This rather simple improvement would considerably decrease the risk of using a bad view. This would be a significant step to continue to allow the users to manage their own view specifications.

6.2 Relations among branches

Although the very general form of branch-types, that can accept versions of files from any other branch-type, might be useful in some cases, the use at KCS shows that branch-types are in practice organized much more structured. Following a very common work process, 'project' branch-types are in fact used to house versions of files from a single 'master' branch-type. The versions of the files on the 'project' are eventually merged back to that same 'master' branch-type. This pattern can be repeated many times, with successive local 'masters' and several such 'projects' active in parallel at any given time.

Supporting such a restricted form of branch-types would have several benefits. It would directly support 'light-weight' branch-types allowing developers in a project to form local short-term projects in parallel with each other. ClearCase would also have the possibility to give an overview

picture of existing such branch-types and how they relate to each other. Furthermore, since these branch-types are restricted, the corresponding view specifications can be much simpler, since they do not have to specify the rules 'all the way up', but inherit the rules of the 'master' branch-type. Together these mechanisms would make it possible for KCS to allow developers to create their own branch-types for local use, although branch-types for new projects still would be meaningful to coordinate.

The suggested 'project' branch-types are similar to the mechanisms in Teamware and would in the same way as for Teamware enable a minimal support for a work process. For the developer this would also mean that he could benefit from using version-control for his own work. The CM-group could with these mechanisms more follow the development rather than act as the central single point of CM-actions.

6.3 Versions of directories

Versions of directories in ClearCase only change when files are added/deleted/rename in the directory. A more ambitious definition would be to create a new version of a directory when some of its files are updated. A version of a directory would uniquely specify the version of each included file as described in [Kat90] and in COOP/Orm [MAM93, MA96]. This much more advanced support for versioned directories would give support for versions of configurations. The possibility to show the difference between versions of the same directory/configuration provides a powerful change traceability mechanism and some of the hard questions when trying to get overview of the development would be easy to answer.

6.4 Awareness

ClearCase, and most other CM systems, are weak on providing 'collaborative awareness', i.e. information on what is happening with the systems due to actions by other developers. In many situations a developer wants to work alone, as if he, or his project members, were the only ones working with the system. In some critical situations, the needs are the opposite, one want to get an overview and immediate information about certain activities. An example we have mentioned earlier is when merging from a development branch-type. The actions of (1) updating the development branch-type with the latest versions on the 'master' branch-type, (2) testing, and (3) merging back could not be interleaved. If interleaving happens, the procedure has to be repeated. Awareness that some project has started such a process might be enough to make other groups hold back, and (on the other hand) noticing that some other group did indeed merge back would be a nicer way to realize that the procedure has to be repeated.

'Awareness' here indicates that the information we request is updated on-line as the changes occur. Aspects of awareness that we see meaningful to support is:

- For a ‘master’ branch-type, that some of its ‘project’ branch-types is in the process of merging.
- Monitoring creation of branch-types in general would be useful for the CM-group.
- For a branch-type, which files currently are changed or checked-out.

7 Future work / further studies

This paper reports on work in progress. Future work will be focused on evaluation of the suggestions put forward in this paper. Since the implementation of the suggestions in most cases would require added functionality in ClearCase, we will most likely try to illustrate the implications of the changes in the COOP/Orm environment. In this environment we plan to do small case studies with some of the source at KCS.

We also plan to work with a prototype implementation in ClearCase for evaluation of our suggestions: a semantic checker for views, support for structured branch-types, versioned directories and awareness through triggers and a client-server architecture. These experiments will have to use existing functionality in ClearCase (triggers, labels etc.) and conventions, so some of the cases might be too slow or cumbersome for professional use, but should be seen more as a proof of concept.

8 Conclusions

The use of ClearCase at KCS has at large been a positive experience. Its facilities has improved the situation for management of its large software system. Although ClearCase has proved very powerful and useful at KCS there are some situations where ClearCase gives less support than one might expect.

The main criticism of ClearCase is the lack of mechanisms for overview, error detection and structure. As a consequence its use needs more training, experience and overview than what most developers have. At KCS the result is that the CM support group has taken on more tasks and the use of ClearCase has been restricted for developers which are not allowed to perform crucial actions. In this way the overview over project and system development has been possible to maintain at least within the CM group. The restricted use also means that the use of ClearCase has been most beneficial for the development managers and less so for the individual developers.

Suggestions for improvements of ClearCase put forward in this paper includes: (1) Better error detection - to enable less experienced users to write specifications themselves. (2) Support for a new kind of ‘branch-type’ to support development projects - enabling more advanced support at integration and merge, better overview of parallel development, and allowing single developers to create local projects. (3) Advanced version control for directories to enable versioning of configurations. (4) ClearCase also has a general weakness in ‘collaborative awareness’. Some information can currently be presented on user demand, while we in this

paper suggest support for monitoring such information. We also ask for more extensive information for developers performing critical tasks to follow certain aspects of changes to the system performed by others.

We feel that with these improvements the deficiencies with ClearCase encountered during its use at KCS would be overcome and the overall usefulness of ClearCase would be much improved.

Acknowledgments

This study has been carried out in close cooperation with the CM-group and developers at KCS. First of all we want to thank KCS for opening their doors for us. We want in particular to thank Krister Erlansson, leader of the CM-group, for the time he has devoted to helping us in this study. NUTEK (The Swedish National Board for Industrial development) has supported this work through grant 93-3564.

References

- [Ask94] Ulf Asklund. Identifying Conflicts During Structural Merge. In *Proceedings of the Nordic Workshop on Programming Environment Research*. Lund, Sweden. June 1-3, 1994.
- [Cla95] Dave St. Clair: Continuus/CM vs. ClearCase, URL: <http://sunsite.icm.edu.pl/sunworldonline/swol-07-1995/swol-07-cm.html>, SunWorld Online, 1995.
- [Clear] <http://www.atria.com/products/clearcase.html>
- [Ced93] Per Cederqvist. Version Management with CVS. Available from Signum Support AB, Linköping, Sweden. 1993.
- [Cont] <http://www.continuous.com>
- [Gus90] A. Gustavsson. *Software Configuration Management in an Integrated Environment*. Licentiate thesis, Lund University, Dept. of Computer Science, Lund, Sweden, 1990.
- [Kat90] Randy H. Katz. Toward a Unified Framework for Version Modelling in Engineering Databases. *ACM Computing Surveys*, 22(4), December 1990.
- [KLMM93] J.L. Knudsen, M. Löfgren, O.L. Madsen, and B. Magnusson, editors. *Object-Oriented Environments - The Mjølner Approach*. Prentice-Hall, 1993.
- [MA95] Boris Magnusson and Ulf Asklund: Collaborative Editing - Distributed and replication of shared versioned objects. Presented at the Workshop on Mobility and Replication, held with ECOOP 95, Aarhus, August 1995. Available as: LU-CS-TR:96-162, Dept. of Computer Science, Lund, Sweden.
- [MA96] Boris Magnusson and Ulf Asklund. Fine Grained Version Control of Configurations in COOP/Orm. In Sommerville, I., editor, *Proceedings of the 6th International Workshop on Software Configuration Management*, LNCS, Springer Verlag, Berlin. 1996
- [MG96] Boris Magnusson and Rachid Guerraoui: Support for Collaborative Object-Oriented Development. In *Proceedings of ISCA International Conference on Parallel and Distributed Computing Systems*, Dijon, Sept. 25-27, 1996, pp 169-174.

- [MAM93] Boris Magnusson, Ulf Asklund, and Sten Minör. Fine-Grained Revision Control for Collaborative Software Development. In *Proceedings of ACM SIGSOFT'93 - Symposium on the Foundations of Software Engineering*, Los Angeles, California, 7-10 December 1993.
- [MM93] Sten Minör and Boris Magnusson. A Model for Semi-(a)Synchronous Collaborative Editing. In *Proceedings of the Third European Conference on Computer Supported Cooperative Work*, Milano, Italy, 1993. Kluwer Academic Publishers.
- [MMA97] Boris Magnusson, Sten Minör and Ulf Asklund: A Model for Semi-(a)Synchronous Collaborative Editing. In *Journal of Computer Supported Collaborative Work*. To appear.
- [MHM⁺90] Boris Magnusson, Görel Hedin, Sten Minör, et al. An Overview of the Mjølner Orm Environment. In J. Bezivin et al., editors, *Proceedings of the 2nd International Conference TOOLS (Technology of Object-Oriented Languages and Systems)*, Paris, June 1990. Angkor.
- [NFS] Sun Microsystems. The NFS Distributed File Service - A White Paper from SunSoft, stb 1252. 1994
- [RBI95] W. Rigg, C. Burrows and P. Ingram: *Ovum Evaluates: Configuration Management Tools*, Ovum Limited, London, 1995
- [Roe75] M. J. Roekind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, December 1975.
- [Team] TeamWare user's guides, Sun Microsystems, 1994.
- [Tic85] Walter F. Tichy. RCS - a system for revision control. *Software Practice and Experience*, 15(7):634–637, July 1985.
- [Tic88] Walter F. Tichy. Tools for software configuration management. In *Proceedings from International Workshop on Software Version and Configuration Control*, Grassau, Germany, February 1988.
- [Wat] Gray Watson. CVS Tutorial. Available from gray.watson@antaire.com.

Distributed Development and Configuration Management

Ulf Asklund

Also published by the Association of Swedish Engineering Industries.
No V040073. ISSN 1493-6444. (in Swedish)

Preface

This report was prepared at the request of the Association of Swedish Engineering Industries (VI). The questions answered in this report were framed by the Technical Council 12 of VI, "System design of software", and of the supportive committee for this study. The project was financed by VI and NUTEK within the research program Complex technical systems.

The study was performed from April 1998 through to February 1999 by Ulf Asklund. It is based upon interviews with people working in industry, studies of research reports and technical literature in the field, and on the collective experiences of the supportive committee and the author. It should be stated that the interviewees have expressed their personal opinions, which do not necessarily represent official company policy.

The objectives of the study were directed by the supportive committee, which also was responsible for the inspection and approval of the final report.

The author wishes to thank the supportive committee and the additional people and companies contributing to this report.

Members of the supportive committee:

| | | |
|------------------|--|---------|
| Annita Persson, | Ericsson Microwave Systems AB | (chair) |
| Ivica Crnkovic, | ABB Automation Products AB | |
| Alf Ek, | Ericsson Mobile Communications AB | |
| Olle Eriksson, | Ericsson Business Consulting Sweden AB | |
| Benita Gadd, | Celsius Tech Systems AB | |
| Jan-Ola Krüger, | Saab AB, Gripen | |
| Thomas Nilsson, | SoftLab AB | |
| Anders Olofsson, | Combitech Software AB | |

| | | |
|----------------|------------------------------|-----------------------------------|
| Jan Rosén, | Volvo IT AB | |
| Ulf Asklund, | Lund Institute of Technology | (project leader/ investigator) |
| Göran Östlund, | VI | (report commissioner) |

Authors address:

Ulf Asklund
Department of Computer Science
Lund Institute of Technology
P.O. Box 118
SE-221 00 Lund
telephone: +46 46 2229641
e-mail: ulf.asklund@cs.lth.se
<http://www.cs.lth.se>

Summary

This report deals with problems in configuration management (CM) related to the distributed development of software. The report is intended for those who are about to, or already have, introduced distributed development. The report contains a theoretical part as well as a survey of how some Swedish companies, who have already started working on these issues, deal with just some of them. Thus the report describes the "state of the art" as well as the "best practice".

Distributed CM has rapidly become important, not least due to the development of the Internet. Therefore, the report may be useful also for those already applying distributed development in any form.

This report focuses on the following areas:

- the work method that should be used to handle distributed development of software,
- different architectures for the organization of computer environments,
- the consequences of the disappearance of informal meetings,
- a brief presentation and characterization of some typical CM tools,
- limitations of present techniques,
- advice for the introduction of tool support

This report gives suggestions for solutions in the presented areas in which problems may occur, and demonstrates that distributed development has the potential to be used efficiently and on a large scale. A considerable proportion of the content of this report was based on interviews at companies. The suggested solutions are practical in nature and are already in use in several Swedish companies. The work methods are therefore well established and are spreading to smaller companies as the infra-structure becomes more readily available and the costs of CM tools are coming down.

Everyone seems to agree that it is always easiest to develop locally. However, it is often not possible to chose and the trend is that an increasing number of companies acquire a distributed environment. However, one trend is that it is becoming increasingly common to create the prerequisite for a distributed development, to be able to make use of the resources within the company in a more efficient way, no matter where they work. Previously, a distributed situation was usually a consequence of a company fusion for example.

Distributed development often results in the utilization of slower and less reliable networks. In addition, it results in a decreased possibility of communication through informal contacts between groups and individuals. Such contacts (e.g. during lunch or in the coffee room) are of great importance in how we deal with unexpected situations.

It is as important to be aware of the methods available and what they are capable of, as it is to be aware of what they are not capable of and what may work less well. Currently, the available methods and tools are accompanied by the limitations of working flexibility in distributed situations. For instance, at present, there are important shortages in the technical support for distributed groups, i.e. for people working closely together, but being situated in different geographical locations. Technically it works better in situations where the members of a group are situated at the same location, whereas different groups may be situated at different locations. However, it is important to be aware of the problems that may be caused by a reduced level of informal communication and information between the groups.

It is important to point out that this report does not give all the answers. It is an active research area and one in which a considerable development can be expected in the forthcoming years.

1 Introduction

Large companies and organizations have for a long time had access to global networks, but the rapid development of the Internet has brought about a dramatically increased access to such services. This results in such a degree of accessibility that it is expected at almost all kinds of work, not least in software development. Groups of developers are now able to work all over the world on the development of the same system. From different locations they may need to modify thousands of different files and sometimes the same files, within a single product. The potential is considerable due to the increased possibility of using personnel and competence in a more efficient, flexible and comfortable manner. At the same time, this new technique has caused considerable changes of the organization of the work place in many other respects, as well as in this one. The way in which the work has been divided and the handling of the interactions between different groups and individuals has been largely affected by the fact that the staff is geographically dispersed. This creates new demands on the tools and the systems used for handling the coordination of the development, especially with concurrent development. Much of these demands, but not all, are within the area of Configuration Management, which is the subject of this report. Other systems for communication and management are of course also affected if the staff are geographically scattered over great distances, but they are not the primary focus here.

Software Configuration Management (SCM) aims at coordinating developers towards a mutual goal - a task that is complicated by the fact that they are geographically dispersed. A central problem is the management the history and development of documents and programs over time as well as the management of branches and to support merge e.g. during concurrent development. SCM was originally developed under the more or less explicit assumption that the people as well as the files are situated at the same geographical location. This applies to the tools that have been developed as well as to the work processes used. The general opinion on the functionality associated with SCM has been formed from this assumption.

Other aspects of system development, such as the creation of a general picture and a context, the communication between developers and groups, have remained manual and without direct tool support. A part of this requirement is covered by individual documents, specifications, and formal meetings where important decisions are made. But an astonishingly large part of this requirement of information for synchronization, within, as well as between groups, is covered by informal contacts between developers, during discussions, at review meetings, during coffee breaks, in the corridor and so on. When people working closely together are geographically disperse we need to consider how these additional aspects may be supported in the work method and in tool support. The closer together people work (despite the geographical distance) and the more dependent their various tasks are, the more apparent the lack of support will be.

A good strategy during all development is to try to limit the dependency between the developers, especially if they are situated at different

locations. This is often already done during the structuring of the product to be developed. The system is divided into modules or components, which are then developed by different groups separately. However, it turns out that despite good structuring, dependencies between the components remain. This becomes clear not least when interfaces need to be modified or when the components are to be integrated. These are examples of situations when one, although one has tried to avoid it, requires an overview and synchronization between the groups mentioned above. In addition, there are problems not directly related to the creation of modules, such as security, change management and the management of original repositories that also require a certain consideration during distributed development.

In this report we demonstrate by using a number of examples, different situations when distributed development may arise in a company. It may be anything from home working, to complete developmental environments at different affiliated companies, or subcontracting (outsourcing). We classify some different cases and highlight their specific characteristics. Furthermore, we describe different architectures, work processes and tools and in what way they support the distribution in each of the various cases. By using company interviews, we have characterized the currently used methods and tools, how they are being used at the different companies and their experiences in doing so. In this manner we demonstrate a "best practice" from the companies. At the end of the report some advice is given on how one does a self evaluation as a first step towards the introduction of the suggested routines and tools into an existing environment.

We also give some insight into the current state of the research within this area and also try to look a little into the future to suggest which solutions may be available as needs increase.

Much of the theory discussed in this report is not limited to software, but may be applied to other products and documents. We will therefore henceforth denote configuration management as CM rather than SCM.

1.1 Objectives

A previous investigation by The Association of Swedish Engineering Industries, "Konfigurationshantering. Motiv och nytta - erfarenheter från svensk industri inom programvaruområdet" [Nym96], considers configuration management in general terms, the focus being basic CM, motives and benefits, particularly the way it has been reported by Swedish industry. For the basic issues regarding CM for software, we would like to refer to the current report. The aim of this new report is to focus on the area of CM in connection with distributed development, an area that has developed considerably since the previous report was written. More specifically, the aim of this report is to:

- study the area of CM for distributed development of software, which is important for the industry,
- supply knowledge in the area of CM to Swedish industry,

- report on national and international progress during the last year in the area of CM,
- follow national and international trends in the area of CM.

This report is primarily intended for CM engineers, project managers, company management and others of an equivalent position.

1.2 Scope

The areas considered, focusing on the distributed development of software, are:

- work methods
- experience
- introduction
- tools

This report is based in part on interviews at 11 Swedish companies with experience in the introduction of distributed development and in part on the available literature and research publications in the area. The interviews were carried out during 1998 and consisted of a one-day discussion followed by one or several iterations to verify, correct and complete the descriptions.

1.3 Reading instructions

This report is not a general introductory description of configuration management (CM). For such a description we refer to [Nym96].

This report is structured such that the introductory chapters (1-5) are theoretical in nature and without evaluation. Terminology that will be used in chapter 6 is introduced here. Hopefully the contents of these chapters will stay the course of time.

Chapter 6 is more practical in nature. Various problems and solutions are discussed and evaluated on the basis what is presently known.

Chapter 2 - Distributed development

Names and describes different cases of distributed development.

Chapter 3 - Configuration management (CM)

Describes configuration management from a management and developer perspective. Terms such as development strategy, strategy for concurrent development and updating strategy are introduced.

Chapter 4 - Synchronization models

A theoretical chapter describing four synchronization models, i.e. different means of supporting the synchronization of simultaneous and concurrent changes by different users. These models are important abstractions

and may, during a discussion, work better than the individual functions of a tool. The summary may be sufficient for a first brief reading.

Chapter 5 - Architectures

Describes different architectures for clients, servers and sites.

Chapter 6 - Experiences and advice on key areas

Contains our definition of some key areas and the problems that geographical dispersion may give rise to. In addition, experiences from the companies that were interviewed as well as specific and practical guidance are discussed.

2 Distributed development

In this report, distributed development refers to the situation arising when developers, or groups of developers, who are developing the same software, are geographically separated. This includes anything from different parts of the same town to different continents with different time zones. As a result, there is a risk of becoming dependent on a slower and less reliable network, and as a consequence having to copy common files - with all the problems that doing this can cause. In addition, geographical separation results in decreased opportunities for meetings, both formal and informal, e.g. at coffee breaks. This means that the informal interaction between groups becomes reduced, resulting in less knowledge of the overall relationship between the sub-projects. It also means that there is a risk that the connectivity within the group (the team spirit) may be weakened at distribution, this complicates the contacts between the groups when there are problems e.g. with a common interface. Thus the situation makes new demands on how to manage common data, e.g. files, but also how information may be spread between and within groups of developers.

The technical problems with distributed development are perhaps easiest demonstrated by the use of a laptop computer. In this case it is common to use the technique of (often manual) *replication* of the file repository. This means that the files that one expects to need are *copied* into the laptop computer. The files are modified on both computers and requires a subsequent synchronization (i.e. the choice of the latest version of the changed files). If individual files have been changed on both computers, they have to be merged. The problem increases with the number of files, the time between synchronizations, and in a more general case, the number of copies.

In addition to the new demands on tools and methods of work for the developers, distributed development also makes new demands on the vertical and horizontal communications within the organization as well as the project management: direction, follow up and reporting.

Some of the problems arising during distributed development can be compensated for by changes in the design of the developed product, improved work models and/or by employing a special functionality of the CM tool being used.

- One may try to eliminate the influence of the new situation by acting as though one does not have distributed development. One way of achieving this is by centralizing the management of updating by having one person responsible for each file, module or sub-system. The advantage of this is that all technical problems with the distribution of the software are eliminated. However, the disadvantage is that the development may become unwieldy, heavy and slow, which may lead to the less important but perhaps simple changes not being accomplished. Furthermore, the person who has the problem is unable to influence which change suggestions are prioritized. Finally, the competence will be less widespread than if for instance a group is responsible, this may lead to problems at staff changes.

- One may try to adapt to the new terms and become less dependent on the aspects that have deteriorated, e.g. make people who are working from geographically separated locations, less dependent on each other. By dividing the common product into components that are as independent as possible, the need for communication between the developers of the individual components is reduced. Despite this, it can occur that they sometimes have to work (at least temporarily) on the same sub-project and then perhaps even with the same files.
- Another way of reducing the need for communication between geographically separate developers is by providing all development locations with a complete copy of all the software. Then the development can be done locally, at least temporarily, which decreases the problems in the daily work. However this solution results in the risk of diverging interfaces causing increased difficulties at integration.

The nature of the work and the distance between the separate users of the CM system (developers, project managers etc.) results in different demands being made on the CM solution. Developers working at home in the evening who have problems, meet their colleagues the following morning and therefore need relatively little communication support from the system. People located in different buildings in the same city may know each other and may therefore get in touch with each other more easily than people located in different countries. Communications over time zones make things even more difficult.

In this chapter we will consider some common situations at which distributed development arise and describe the characteristics for each of them. Later in the report we will also classify some different client/server architectures and examine their advantages and disadvantages regarding distributed development.

Distributed development affects more than just the architecture of the application. For instance, change management, management of original repositories, implementation of incremental development and so on, are also affected. A more detailed discussion of these areas is given in chapter 6, “Experiences and advice on key areas”.

2.1 Cases of distributed development

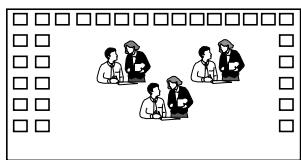
Distributed development may arise due to several different reasons. In this chapter we will try to classify these reasons using some characteristic cases. We hope that the reader may recognize one or more of the cases, indicating that distributed development has already been, or is about to be, introduced. A classification also facilitates a discussion regarding suggestions of solutions as well as later discussions in this report. The different cases that have been identified are:

- Locally (for comparison)
- Distance working

- Outsourcing
- Co-located groups
- Distributed groups

The different cases occur individually or in combinations. For instance there may be groups which are normally connected but which may occasionally be distributed.

Locally

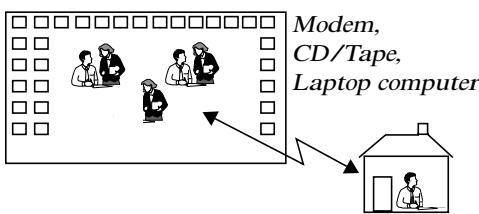


meetings as well as by more informal encounters such as at the coffee table. Informal meetings also create a team spirit, which in turn increases the probability that the established CM process is observed.

From a CM perspective:

- A common file system.
- Complete development and test environment.
- Synchronization can to a certain degree be achieved through meetings. In particular, problems that arise can be solved through direct communication.
- Good awareness of what others are doing (group awareness).
- No particular security problems (external networks are virtually unused).

Distance working



basis or for longer periods of time, a situation similar to that for distributed groups arises.

A limited computer utility and a relatively slow means of communication with the world around (for instance by data modems to the usual place of work) is characteristic of distance working. Despite this, there is a desire to be able to start working quickly, as the total working time on each occasion is short (typically a few hours in the evening), which means that it must be possible to set up the working environment quickly. As the daily contacts remain, the possibility of informal communication and

A fast network is characteristic of a place of work where everyone is situated locally, allowing complete development and test environments for all developers. It is fairly easy for the project groups to communicate and synchronize their work, by formal

meetings as well as by more informal encounters such as at the coffee table. Informal meetings also create a team spirit, which in turn increases the probability that the established CM process is observed.

From a CM perspective:

- A common file system.
- Complete development and test environment.
- Synchronization can to a certain degree be achieved through meetings. In particular, problems that arise can be solved through direct communication.
- Good awareness of what others are doing (group awareness).
- No particular security problems (external networks are virtually unused).

This kind of distant work is brief work being performed elsewhere than the usual place of work. Home working as a complement to the daily work being the primary example.

When developers work at home (or elsewhere) on a more regular

maintaining the team spirit is more or less the same as in the local situation.

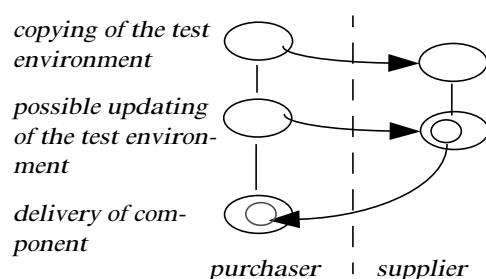
Two common modes of working are:

- Individual files are brought home and worked on locally. This is a necessary working mode when a complete environment at home either takes too long to update at each occasion or cannot be installed.
- Remote login to the place of work and the home computer is being used as a terminal.

From a CM perspective:

- Bringing home individual files results in the work being done locally outside of the control and support of the CM system. The degree of impairment this can lead to partially depends on which synchronization model the tool supports, see chapter 4, "Synchronization models". For instance no support is offered as to the awareness of what others are doing simultaneously. In addition, testing is made impossible.
- Login at a terminal is similar to the local case. The slower connection makes the work somewhat heavier going for the developers. Then there is also a tendency that they may not follow the work models the way they should (for instance to make a complete test of all platforms before checkin).

Outsourcing



Instead of developing everything by yourself or buying existing components (COTS - Commercial Off The Shelf) you may have a third party develop them for you. This is usually called outsourcing (or subcontracting) and gives, compared to COTS, a greater control of the development of the component,

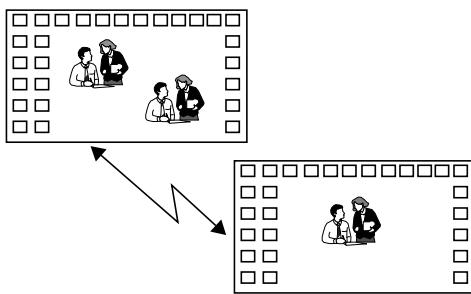
albeit at a higher price. Outsourcing is based on a close collaboration between the supplier and the purchaser. Consequently it is often possible for the developer/supplier to test the component in an environment similar to the target environment prior to delivery. The purchaser then usually provides the test environment.

The purchaser is ultimately responsible for the product and possible error/change management can be reflected in changed demands on the component towards the supplier. As with any order, it must be clear what should be delivered, but in this case it is further complicated by the fact that the demands as well as the environment may change.

From a CM perspective:

- The purchaser must be able to integrate new versions of the component into the product, which itself may have developed since the latest release of the component.
- The supplier should be able to manage the updating of the development and test environments.
- The purchaser and the supplier do not necessarily have the same CM tools, which might make the updating (in both directions) difficult.
- At delivery of a source code, the generation tools must also be consistent between the purchaser and supplier.
- With changed demands, the connection between the version of the demand and delivery must be clear.

Co-located groups



Developers at different affiliated companies usually belong to local groups or projects. The division of the work has already been determined at the structuring of the project/product to prevent too much dependency between the different groups. The product is divided up into sub-products, which can be

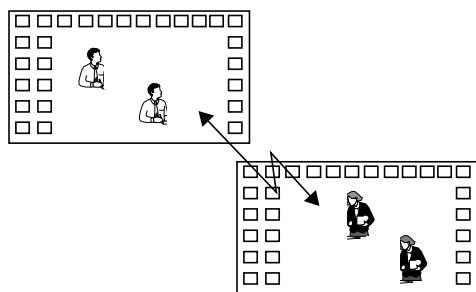
developed by different project groups. The division makes it possible to do most of the development locally within the groups without the requirement for much communication with other groups. Within the group and between groups in the same place, the situation is the same as with local development. Groups in different places normally only have access to the latest stable versions produced by the other groups. Due to the geographical distance, potential problems will inevitably be more difficult to solve. Therefore, updating and distribution between the groups requires more effort and administration, these may be considered as internal deliveries and therefore tend to come more infrequently. Cooperation between the groups may be facilitated if the work is planned in phases of which everyone is aware. Conversely the redistribution or division of the work is more difficult to perform afterwards.

From a CM perspective:

- The files are stored in different file systems, but (ideally) in the same CM system. Large companies sometimes have different CM systems in their different affiliated companies.
- When the locations are permanent, each local group should be able to work within a complete development environment and with the possibility of testing.
- The groups deliver (release) sub-products between them rather than develop together.
- There are often few or no unplanned daily contacts between the groups. The contact is limited to e.g. weekly meetings, which may be actual physical meetings or telephone/video conferences.

- It is important to maintain the knowledge of the development status between the groups.
- Change management of common components, such as interfaces, is of particular importance.

Distributed groups



Distributed groups with members at different locations means that the members of the group are also distributed, i.e. that the people working in the same project, perhaps even in the same files, are geographically dispersed. The possibility of daily communication by formal as well as informal meetings is lost even *within* the group.

Projects working towards the same product usually use some common libraries or components. Changes in these are unusual (simply because they are common and changes are difficult to manage), but sometimes inevitable. If group members at different places want to make changes at the same time they face a situation similar to that for the updating of interfaces where there are “connected groups” but in this case the problems apply to all files.

The situation with distributed groups can usually be avoided, by considering separate individuals as very small connected groups for example. Despite these efforts, there are cases when the groups need to work more closely together although they are still distributed. The obvious example is when people included in one group, have to travel around to other groups for various reasons. Of course there is a desire to be able to continue working with the usual project, this will then be done as a distributed group. A similar situation arises when staff are moved to new projects but often need to be consulted on the old project. People with special competence are often included in several groups, which can be at different locations.

From a CM perspective:

- It is important that the members of the group receive information about what the others in the group are doing, how the project is developing, its status, which changes have been done and by whom etc.
- It is important to support the division of files and concurrent, simultaneous changes.
- Solutions using “locking” and exclusive access to files work poorly as it is difficult to solve situations where group members, located at different sites, must wait for each other.

Discussion

The situation of local development is of course preferable from a CM point of view, as it is easier to manage than the cases of distributed develop-

ment. However, there are several other good reasons for the use of the different situations outlined above.

The situation with connected groups usually results in the work being planned in a manner such that the dependency between groups in different places is minimized.

The situation with distributed groups is usually not desirable, but rather the planning of the work, the complete system construction, the division into components and so on aims to avoid this. However, it can be anticipated that such a situation arises as a consequence of the break up of connected groups.

An additional example is in using remote places of work, i.e. a place of work situated closer to home than the “real” place of work, which is therefore used most of the week. The situation is a combination of distance working and distributed groups. Typically, formal meetings work, but informal ones, either partially or completely, fail to occur.

3 Configuration management (CM)

Before entering deeply into the distribution aspects we will give a short introduction to CM in general. For a more extensive introduction we refer to the earlier report from The Association of Swedish Engineering Industries, "Konfigurationshantering. Motiv och nytta - erfarenheter från svensk industri inom programvaruområdet" [Nym96]. Other good references considering various perspectives of CM are [TF94, App98, Whi91, Kel96].

The definition of CM given in the previous report is:

CM is the controlled way of leading and managing the development of and changes of combined systems and products, during their entire life cycle.

However, there are several other definitions, all with a different focus. One reason for the differing definitions is that CM has two target groups with rather different needs: management and developers.

From a *management perspective*, CM directs and controls the development of a product by the identification of the product components and control of their continuous changes. The goal is to document the composition and status of a defined product and its components, as well as to publish this such that the correct working basis is being used and that the right product composition is being made. One example of a definition supporting this discipline is ISO 10 007 [ISO95] meaning that the major goal within CM is "to document and provide full visibility of the product's present configuration and on the status of achievement of its physical and functional requirements".

From a *developer perspective* (tool support), CM maintains the products current components, stores their history, offers a stable development environment and coordinates simultaneous changes in the product. The goal is to make a group of developers as efficient as possible in their common work with the product. CM includes both the product (configuration) and the work mode (methods). The definition by Babich [Bab86] stresses the fact that it is often a group of developers that shall together develop and support a system: "Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team".

3.1 Strategies/working modes

A superior aspect for the design of CM is what *development strategy* is being chosen when modifications of a system are to be made. The strategy must take into account two basically contradictory desires. On one hand one wants to bring about the early integration of changes such that potential problems are discovered as quickly as possible. On the other hand one wishes to give the developers a stable environment to work in, such that they will not be unnecessarily disturbed in their work by the occasional faults of others. Strategies emphasizing the first desire may be considered

as *optimistic* - the problems may not be that great, and strategies emphasizing the second may be considered *conservative* - requiring a great deal of integration work, which is complicated, and so it is done as seldom as possible. An example of an optimistic strategy is “Daily build” where all of the day’s changes are integrated at the end of that day. The one integrating last will have to deal with the problems. An example of a conservative strategy is “Big-bang integration” which tends to be done late, close to the release, and then involves all of the developers over a rather long and intense period of time.

A related aspect is the strategy for how to allow *concurrent development*. A previously common conservative strategy is to not allow concurrent changes in the same files (or even sub-systems). An optimistic strategy is to allow such changes (more or less planned for) and to accept that a later integration with a merge of the changes may require a certain degree of work.

We may also talk about the *update strategy*, i.e. (a) how one makes changes available to others, publishes them, and (b) who ensures that the changes are actually being used, subscribed to. An optimistic update strategy is to have all changes that are published, immediately being used by others. This means that all integration problems have to be solved immediately (although one tries to avoid these problems by an extensive process of code inspection and testing before publication). A conservative update strategy means that published changes do not immediately take effect but the one subscribing to the changes can control when this will happen, which means that it may be later (but at the earliest when they discover the change).

The attitude to concurrent work in the same files/sub-systems has, in several cases developed from a conservative to an increasingly optimistic attitude as tools for version control and merging have come in general use. In distributed development, the possibility of doing concurrent development at least to some extent, is more or less a prerequisite.

An optimistic update strategy can be combined with a conservative development strategy. When in a big-bang integration, after a thorough analysis, one thinks that one has characterized all of the integration problems and defined an order in which the changes should be integrated, one immediately wants to discover if additional and unexpected problems arise. Conversely, a conservative update strategy may employ an optimistic update strategy. For instance, in “daily build”, one integrates towards a common line of development, but the changes will not affect the other developers until they start their own integration. This and other incremental development strategies are optimistic as they frequently use the integration phase.

Irrespective of which strategy is best suited to any given situation, it will always affect the development model being used, make demands on the tools being used and of course also make demands on how the CM work is being structured. For instance, an optimistic update strategy may result in the model including extensive quality assurance e.g. by code inspection and tests *prior* to publication as well as that the CM work, e.g. documentation of changes, is planned such that tracking the changes and system backup are facilitated.

3.2 CM from a management perspective

CM is a broad discipline covering the entire development process, both in the time (the entire lifetime of the product) and in the extent (product and process). In this chapter we take a management perspective and focus on what CM means from this point of view. It is not without reason, that CM is already a key area for level 2 to achieve repeatability according to the known CMM model [SEI95].

Areas of responsibility within Configuration Management.

Extract from ISO 10007 [ISO95]

- **Configuration Identification**

Activities comprising determination of the product structure, selection of configuration items, documenting the configuration item's physical and functional characteristics including interfaces and subsequent changes, and allocating identification characters or numbers to the configuration items and their documents.

- **Configuration Control**

Activities comprising the control of changes to a configuration item after formal establishment of its configuration documents.

Control includes evaluation, coordination, approval or disapproval, and implementation of changes. Implementation of changes includes engineering changes and deviations, and waivers with impact on the configuration.

- **Configuration Status Accounting**

Formalized recording and reporting of the established configuration documents, the status of proposed changes and the status of the implementation of approved changes.

Status accounting should provide the information on all configurations and all deviations from the specified basic configurations. In this way the tracking of changes compared to the basic configuration is made possible.

- **Configuration Audit**

Examination to determine whether a configuration item conforms to its configuration documents.

Functional configuration audit: a formal evaluation to verify that a configuration item has achieved the performance characteristics and functions defined in its configuration document.

Physical configuration audit: a formal evaluation to verify the conformity between the actual produced configuration item and the configuration according to the configuration documents.

ISO95 also describes how to document and establish the work process in a special CM plan: *A configuration management plan (CMP) exists for application within the organization, for projects or for contractual reasons.*

A CMP provides for each project the CM procedures that are to be used, and states who will undertake these and when. In a multilevel contract situation, the CMP of the lead contractor will usually be the main plan. Any subcontractors should prepare their own plan, which may be published as a stand-alone document or be included with that of the lead contractor. The customer should also prepare a CMP that describes the customer involvement in the lead contractor's CM activities. It is essential that all such plans be compatible and that they describe a CM system that, will provide a basis for the practice of CM during later project phases.

As seen by these definitions, the purpose of CM is to organize the work at the software development level, i.e. the software development process, as well as on the product level, i.e. at delivery (external or internal). The purpose of the CM plan is to make the routines clear and known and of course adjusted such that they are easy to work by. The effects of geographical separation are already included in these definitions, for example in the discussion of supplier and purchaser in the CM plan, however, regarding software development, the four points are written without such considerations. This does not prevent the formulation of a concrete CM plan being considerably affected by the fact that the work is to be carried out in a distributed environment.

- Configuration identification – In distributed development, it is even more important to create a component structure enabling as independent development of the components as possible, than in an entirely local development.
- Configuration control – In this paragraph change management and who approves of changes is discussed. Should this function also be distributed or is it best when managed centrally? How does one manage changes affecting the development at several sites? How and who prioritizes which changes should be implemented and in which order?
- Configuration status accounting – Is there a risk of loosing the complete picture if the follow up is only done locally? Who can make decisions regarding the reprioritization of tasks carried out at different places? How does one get a complete picture of the status - is this the same thing as the sum of the parts, or does one then miss something essential?
- Configuration audit – In distributed development, one can largely consider the completion of components as deliveries, and in management terms, compare distributed groups to sub-contractors. Then perhaps deliveries and therefore audits will occur more often. If so, does that now imply that there are too much "overhead" and how does one avoid all the integration work being done by the purchaser, i.e. centrally?

Thus CM from a management perspective is affected in several aspects by the fact that the work is done in a distributed manner and that the mode of work and the development process used, will in many cases probably have to be revised. Development is often stepwise and for a first adaptation, one may use the experience one has from the handling of subcontrac-

tors. As the distributed development becomes a greater and more natural part of the activity, and as one gets more experience, the CM handling can gradually be adjusted. Several companies have come a part of the way, and their efforts and experiences, which are reported in chapter 6, "Experiences and advice on key areas", can serve as a source of inspiration and guidance.

3.3 CM from a developmental perspective - tool support

We have described the realization of a development strategy in a plan for the management and documentation for the development and changes of a system and a procedure, an order, for how and when changes can be implemented and integrated. From the developer's point of view, much of this work may be considerably facilitated by the use of suitable tools in the daily work.

The CM-related functionality one may wish for is extensive and does not diminish, therefore we also consider the problems in connection with distributed development. In this report we mainly focus on the tool aspects that we regard to be most relevant in this context. These are as follows:

- Version Control - the possibility to store different versions and variants of a document and to subsequently be able to retrieve and compare them.
- Configurations/Selection - functionalities to create or select, associated versions (or branches) of different documents.
- Concurrency Control - manages the simultaneous access by several users, i.e. concurrent development, either by preventing it or by supporting it.

These aspects are expanded on in further detail below. However, firstly we will briefly discuss other CM-related functionalities, which may also be relevant for tool support and where distributed development may play a role, but which will not be discussed further in this report. Some of the aspects involve a terminology, which is often referred to, and is therefore introduced here:

- Build – mechanisms for keeping generated files up to date, for instance during compiling and linking, preferably without generating anything unnecessarily. In a distributed development situation, it is possible that the build result is divided, but in many cases it is also likely that it is maintained where it is required.
- Reporting, status – the reporting of a current status with lists of which files have been changed during a certain time period, who made the changes, differences between products etc. These are important functions particularly in the support of the overall view,

as seen by the project management. They are probably even more important for distributed development, but despite this we do not regard them as crucial when examining the models and tools.

- Process support – tools that help the developer follow the development model and perform the actions prescribed by it and the CM plan. The value of such automated tools may be even greater in distributed development if the tasks can be moved between different locations.
- Change management – a system supporting the management of the collection of change requests, for instance in collaboration with customer support (“Help-desk”), the generation of error reports, firm change requests, implementation of those changes, documentation of the problem and the solution, and when it is available. Much of this work is already distributed nowadays, by the fact that the organizations for support and development are separate. In distributed development, the situation is not so very different, although an integrated tool support would offer an increased tracking ability (problem - change request - changes).
- PDM - Product Data Management – the management of the product structure including components, software and documentation. PDM, in principle, does not manage software because the support systems lack the functionality to build for example. This is a whole area in itself and is not dealt with any further in this report.
- Accessibility Control (Security) – preventing inappropriate access to information without complicating normal work. This is a highly relevant problem in a distributed development situation.

Version control

The possibility to store, recreate and register the historical development of a document or file is a fundamental characteristic of a CM system. Every stable issue of a file's content is termed a *version*. Versions of a file may be organized in a number of different ways. When organized in a sequence they are called *revisions*. They may also be organized as parallel development lines called *branches*. Branches can be merged into a new version, which then has two or more predecessors.

Revisions are usually deliberately created by a developer, e.g. when (one considers that) a task is completed. In addition, many editors maintain one or several micro-revisions of a file to facilitate its recovery following unsuccessful editing.

Branches are created for several reasons. The primary ones being *permanent branches*, these adjust the file according to diverging demands for instance different operating or window systems, and *temporary branches*, to permit parallel (concurrent) work. In the latter case, the branches are merged when the reasons for the concurrent work disappear. Usually, a branch consists of a series of revisions and additional branches can be created from the original branches etc. Branches are created for a reason and are therefore not considered to be equal but to play different roles, for

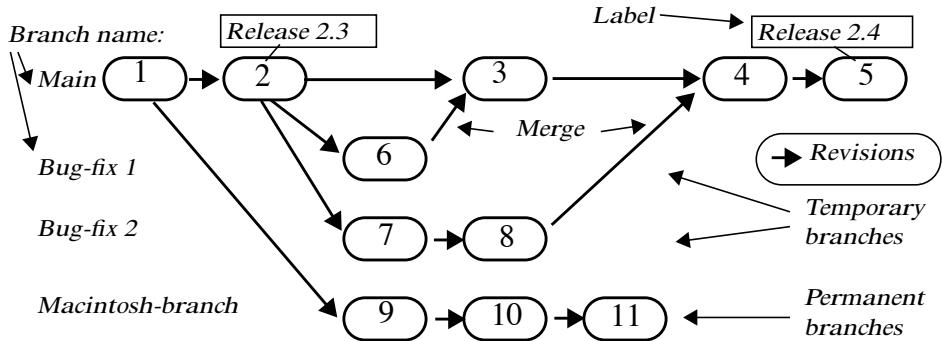


Figure 1 Basic version control. The versions (1,2,3,4,5) are revisions in the same branch ('Main'). Version 2 has been named "Release 2.3", whereas version 5 has been named "Release 2.4". "Bug-fix 1 and 2" are temporary branches (with one and two revisions respectively) which have been merged back to "Main". The example also illustrates a permanent branch with three revisions (9,10,11).

example, as the main line in the development process or as a branch for the implementation of a change, a bug-fix.

When whole products or configurations are adjusted according to diverging demands, this is managed with *variants*. For instance, different variants of a product may be developed for different operating systems or with different customer adaptations. The creation and maintenance of these variants can be done in three ways:

- with permanent branches of the included files. For a variant, file versions are primarily selected from the permanent branch created for the purpose. Secondly, a file version from a variant independent branch, e.g. Main, is selected.
- with conditional compiling (compiling directives). This means that all variants are managed in the same version of the file and are therefore easier to keep together. However the variant management will not be visible at the CM level.
- with installation descriptions clarifying which functionality should be included in a certain variant. Variant dependent functionalities are implemented in different files, one for each variant.

A tool for version control can internally identify revisions, usually utilizing a numbering technique in several stages that may be user friendly to a greater or lesser degree. In addition, the user themselves can usually give the revision one (or several) optional names in the form of a string, a tag or a label. The tool can return a version identified by such a string. This facility ("tagging") can be used to realize a simple selection mechanism as illustrated in Figure 1.

Configurations/Selections

As shown above, there are often a great number of file versions, and which one should be used in a given situation is not always obvious. The situation is further complicated by the fact that a system consists of a large number of files such that the possible number of combinations is

enormous. In a development situation, for files being worked on and for which a special temporary branch has been created, one usually wishes to have the latest revision in that branch. For other files, one typically wants an older, stable version, for instance one that is included in the latest release, or the most recently published stable version as developed by other groups. For files developed within ones own group there is a great need for flexibility, such that it is possible to control how close one wants to be to the others development. Several change requests require the modification of more than one file. In all situations, it is desirable to ensure that there is a consistent selection and configuration, in terms of the inclusion of versions with connected modifications.

A useful technique for the specification of a configuration supported by several systems, is to offer a rule-based selection mechanism. Typical examples of rules that one would like to be able to specify are:

- the latest revision in my own branch (for files that I myself /the group work with),
- the latest revision in a named temporary branch (for files that other groups work with),
- the latest revision in a named permanent branch (for files that differ depending on the product),
- fixed, named, version, e.g. the latest release (for other sub-systems).

A system being built using a rule specifying the “latest” is called a *partially bound* (sometimes “generic”) *configuration*, as the exact versions that are included, will vary in time. A system being built without such a rule is called a *bound configuration* and is particularly suitable for deliveries, as the versions of all files included are fixed and therefore it can be guaranteed that the system can be recreated. The difference is also connected with the discussion on strategies above (chapter 3.1, “Strategies/working modes”). Rules giving partially bound configurations, permit an optimistic update strategy as a newly made revision in the corresponding branch takes immediate effect (at the next build) without the person building the system having made any changes.

Certain bound configurations can form a baseline, i.e. are a basis for further development with formal change management.

In the same way that the development of individual files can be considered to be a version history, so can a corresponding development of configurations. As an example, the user/customer sees the development of a system in large steps, namely the configurations, releases, that are distributed. Developers and project managers see many more stages in the development of the system and also the division into sub-systems and configurations, with their own version histories. Therefore the perspective where a system and sub-system are regarded as the development of configurations in bound configurations may be useful at several levels.

The facility of naming versions (“tagging”) can be used to manage the selection of bound configurations in that all files are tagged with the same name, e.g. “Release 2.3”. Relations between such configurations, for instance that “Release 2.3” is based on “Release 2.2”, will not be supported

by the tool but have to be managed in a different manner, for instance in a log.

Consistent naming may also be used to represent *logical changes*, i.e. changes arising from a *change request* and result in the modification of *several files*.

Concurrent development - synchronization and “collaborative awareness”

All software development employing a large number of developers is carried out as concurrent work, as is distributed development. The work is usually carried out in groups that are separated to a greater or lesser degree and their contributions must therefore be coordinated, synchronized. Thus concurrency control is not only required between groups but also within a group.

The formulation of tasks, for instance those issued as result of a change request, are mainly a management function, the goal usually being to formulate these change requests such that they can be carried out independently of each other, simultaneously, and as concurrent as possible. As previously stated, the independence becomes even more important if the groups who will perform the changes are distributed. Despite these efforts, it can occur that different groups need to modify the same files and thus the synchronization of the groups involved becomes essential. We have previously described different strategies for concurrent development as conservative and optimistic. An example of a conservative strategy is locking: – where only one group is allowed to modify a file at any one time. An optimistic strategy would be to allow simultaneous concurrent modifications and offer support for a subsequent merge of the two modifications.

Locking can in practice, only be used within a small group without encountering awkward blockades between developers and groups. With distributed development it is even more problematic for several reasons. It is technically problematic to guarantee locking in a distributed environment where networks may be down. Locking may also lead to blockades between developers. Given all of the above, and although it is impractical, the problems can be solved in a local environment by direct interaction between the developers who themselves decide who shall make their changes first. In a distributed environment this is difficult for many reasons, time differences, less social contact and perhaps language and cultural differences. In this environment with a large number of developers, it can be suspected that such problems will occur frequently.

Temporary branches, which were previously described (chapter “Version control”), are intended to be used for the management of simultaneous concurrent modifications of a file that arise on employment of the optimistic strategy. Here, the problem of who will perform the merge of the changes, the test and the integration of the result, arises instead. As we shall see in the next chapter, this problem is directly connected to the selection of an optimistic or conservative update strategy.

Another important aspect of development where there are several developers involved, is how they maintain their orientation and awareness of the overall development, such as what other developers or groups

are doing and their status (“collaborative awareness”). Part of this orientation is through formal channels, via project managers, meetings and so on, but a very large part of such information and experience is spread through informal means. This may occur in the coffee room, over lunch, in the corridor or in their spare time. Experience from other computer aided distributed applications (“groupware”) has shown that this kind of information distribution, which is complicated in a geographically distributed development situation, is very important. It plays a large role in ensuring that groups avoid creating problems for each other and that they understand and solve problems when they eventually arise. It is very important to try and replace this means of information distribution on the introduction of distributed development. At the same time, existing CM systems are weak when it comes to offering such information to developers and project managers, and other means of making such information available should also be considered. One example may be the availability of easily accessible information on the status of a sub-project, which files are being changed/have been changed and by whom etc. Systems for an exchange of experiences, e.g. FAQ (“Frequently Asked Questions”) may also serve such a need.

3.4 Summary

CM has two target groups, developers and project management, that both have a somewhat different focus. This contributes to the feeling that CM is often regarded as a diversified discipline with different goals and scope.

From a management perspective, we have identified four areas of responsibility: configuration identification, configuration control, configuration status accounting, and configuration audit, as well as the requirement for a plan of the CM work.

From a development perspective, for distribution, we have identified three particularly important areas: version control, configuration management and concurrent work.

In addition, we have determined that there are some superior strategies that have to be considered when defining a CM plan:

- Development strategy - product integration, often/rarely
- Concurrent work - changes of common files
- Update strategy - when/how are modifications available for other developers.

In all cases, we characterize different strategies as either optimistic or conservative. These fundamental starting points are important for the understanding of the following chapters.

For distributed development situations, we have also emphasized the need of replacing the informal forums for contacts and interaction with some other mechanism of “collaborative awareness”.

Taking the work mode/intended work mode (formulated as concrete strategies according to the above), the CM plan being constructed and the process model intended for use as a starting point, it can be decided which

of the synchronization models (to be introduced in the following chapter) may be useful. One can also be guided by the chapter summarizing various experiences.

4 Synchronization models

As an important aid for the developers, all CM tools offer some kind of support for the synchronization of simultaneous, concurrent changes from different users. Depending on the tool, this support is given in different ways according to different synchronization models:

- Checkout/checkin. Concurrent development by temporary branches of individual files/objects.
- Composition model. Configurations are represented by selection rules. Developers work concurrently with different selection rules and files in their own working area.
- Long transactions. A system/configuration is developed as a series of versions, which may include changes to many files in the configuration. The coordination is achieved when variants of the configurations (concurrently developed) are integrated (merged), this can mean that several included files have to be merged in turn.
- Change set model. Keeps the configuration management picture focused on logical changes rather than on files or configurations.

Most tools support one or two of these models, usually checkout/checkin plus one of the more configuration based models. On the selection of a tool, it is important to select one supporting the update strategy being used. For a developer to obtain full support from the tool, it is important to understand and utilize the synchronization model supported by the tool, otherwise the tool could easily be regarded as an obstacle rather than a support.

4.1 Checkout/checkin

The first generation of tools, for instance SCCS [Roe75] and RCS [Tic85] were entirely designed for use with a local file system and focused on individual files. They have been used extensively together with build tools such as make [Fel79] either directly or indirectly via other tools that have been built on top of them. The model supported by them is checkout/checkin, where individual files are stored in a compact form on a version control basis, in a small database, a repository. Files are not read or changed directly in the repository without being checked out first. Checkout means that the file is copied into the developers working directory and if write access is required, the file is “locked” in the repository. Locking prevents other developers from checking out that particular file (or more specifically that branch, see below) in write mode. When the file is checked in, a new revision of the file is created in the repository and the lock is released. In this way, each file in the repository will get its own version history with a new version for each checkout/checkin.

These tools support “tagging”, i.e. the technique where named versions and bound configurations can be represented by the sequential naming of all files involved. Figure 2 illustrates the users local development environment with checked out files, editor and generation tools as well as the

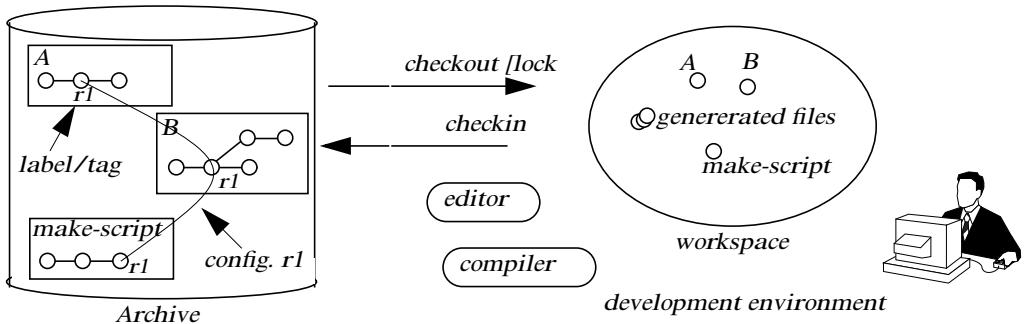


Figure 2 Repository and workspace in the checkout/checkin model

repository protocol. It should be appreciated that in the repository, there is a bound configuration, “r1”, from which all files have a version marked with this name.

The files that a developer does not need to modify but only read/compile, can be managed in two ways. The developer can copy them from the repository and save the copy, in this case a conservative update strategy is obtained. If subsequent revisions of these files are added, they will not affect the developer's system until he again decides to retrieve files from the repository. Alternatively, they may be read from the repository each time they are required, in which case, other developers changes are implemented as soon as they have been checked in, in this case an optimistic update strategy is obtained.

Comments

The advantages of the checkout/checkin model are that it is easy to understand and due to its simplicity, is also included as a part of several other models. The developers can work with their tools and directories as usual and consider CM as a repository function.

However the checkout/checkin model is very simple and has a number of drawbacks that the newer models try to solve with more supplementary functionality:

- Bound configurations can be represented by “tagging”, i.e. by named versions of the files included. However, this does not support the configuration version history or the relationship between configurations. The users have to rely on convention on how to create names and other documentation (which are not understood by the tool either).
- No support for logical changes, i.e. change requests requiring the modification of several files. These have to be checked out/in separately. The ability to track simultaneous changes (from the same change request) is therefore not supported.
- Poor integration. The tool (e.g. RCS) manages the repository and then focuses on the version and the branch management of individual files. Build tools (e.g. make), editors and so on work in the file system and local working directory completely unaware of other

versions. For example, the build tool has the description of the configuration without knowing anything about the version control. There is a possibility of letting the build tool check out the versions to be read.

- The model is dependent on the file systems facilities for access rights. It is the file system that must prevent that files being checked out to be read only are not modified. It is very easy for the initiated user to bypass these protections without leaving a trace.
- Poor support for version selection. Each file has its own version history and the version history of the system (or a configuration) is difficult to overview. The technique of “tagging” configurations is possible in principle but as all files (not only those that have been changed) have to be “tagged” this soon becomes a complicated process.
- Flat structure of the database. To keep a directory structure a repository must be inserted into each directory.

When the number of developers is high, particularly with distributed development, the following points become more obvious:

- Locking of files does not work well when there are several developers. A locked file can not be changed by anyone else and the “work-around” is to create a new temporary branch in which one works. However, it is not possible to integrate changes in the original branch until the first user checks in. As the selection rules of others are often “the latest from the main branch” they cannot, at least in an easy way, utilize changes which have not been checked in there. In addition, the other user must monitor the situation to be ready to check in whenever possible.
- The model often results in long checkout times. This is due to the fact that it is often used with an optimistic update strategy, which means that by checking in changes, others can use them. This is not desirable until the complete sub-project is stable, this means that the individual developer checks in as infrequently as possible. The developers do not see any particular advantage for the use of version control in their daily work and also cannot see what has been changed as the repository is rarely updated.

4.2 Composition

The composition model is a further development of the checkout/checkin model. The extension consists of a better-developed support for the management of configurations and thus different strategies for updating between developers. On the other hand, repository facilities, checkin and checkout, working directories and the synchronization of simultaneous changes using the locking of individual files are the same as in the check-out/checkin model.

The definition of a configuration is made in two steps: (1) a *system model* selects the components that shall be included in the configuration

and (2) *version selection rules* then determine the version of each component. This is graphically shown in Figure 3 in the form of an AND/OR graph, with step 1 representing AND and step 2 OR. The method works recursively down the hierarchy of configurations until all components and files that should be included have been selected for a specific version. Due to the fact that the selection process can be started at any level (not necessarily for a whole system) the CM system can manage system families at different levels. A more detailed description of the AND/OR graph is given in [Tic88].

At checkout, the system model and selection rules are used to determine which file and which version one will have. At checkout for making changes, one copy of the file is placed in the working directory. The changes then done are therefore outside the control and support of the CM system. Checkin often occurs in two steps. The first is in what is regarded as a local configuration accessible to the developer (or the local group). Later, the local configuration is made accessible to other groups, often after a merge with changes made by other groups. This itself can be done in several steps.

“Tagging” can be used to label each respective version of the files included in a bound configuration. Such labels can be subsequently used in the selection rules for configurations where one wishes to start from later work, for example an existing release. The update strategy may be affected by the selection rules:

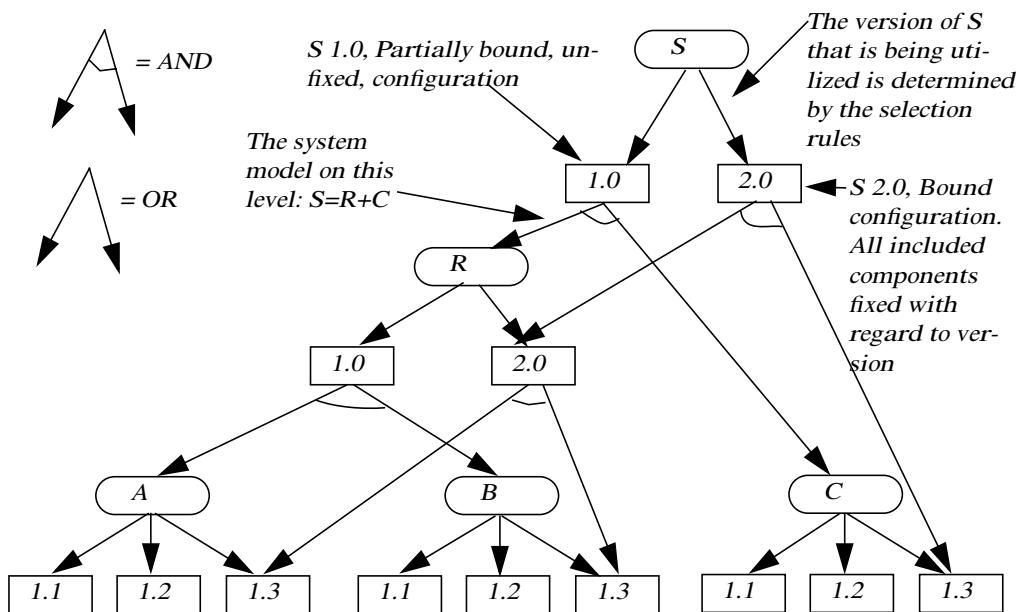


Figure 3 AND/OR graph representing several versions of the system S . The system model for S is $R + C$, and for R it is $A + B$. S in ver 1.0 is in a partially bound configuration whereas for example the version of C is dynamically determined. S in ver 2.0 is in a bound configuration where all included components are fixed to the version, e.g. version 1.3 of C .

- To obtain an isolated workspace which is not affected by the work of other developers, a configuration utilizing selection rules giving fixed versions of all the files except those that the developer himself has checked out and is therefore changing, is defined. The developer himself is then able to decide when to integrate the changes made by others by then changing the selection rules, i.e. a *conservative update strategy*.
- By using generic rules such as “the latest checked in version”, one gets a configuration that changes as other developers check in their modified files. The changes then take effect immediately (at the next build) and one obtains an *optimistic update strategy*.
- The closest form of collaboration is obtained by the use of shared workspaces, when several developers use the same system model and selection rules. In this environment, developers also share modified source code files as well as generated files (if wished), as in the simple checkout/checkin model. Workspaces can, in practicality, only be used in a local environment.

The technique of using selection rules in the Composition model results in a stronger support for defining configurations than checkout/checkin does. The rules make it possible to think and work in configurations, rather than in individual files. However, configurations arise indirectly as a result of rule evaluation and no direct support for version configuration exists. By version controlling the system descriptions and selection rules, one can get some support for giving a version to the configuration. For the representation of bound configurations the only option is to use the technique of named versions (“tagging”).

Comments

The Composition model can be considered to be an extension of the checkout/checkin model. This extension consists partly of selection rules and partly that one can manage the configurations (often in the form of directories in the file system). Thus the model shares many of the drawbacks previously listed for the checkout/checkin model. These include insufficient support for setting versions for configurations and the tracking ability. The model also gives insufficient support for the awareness of activities between developers and groups (“collaborative awareness”). This stems from the fact that the actual development work is carried out in workspaces not managed by the CM tool. If using an optimistic update strategy, this often results in a conservative development strategy, i.e. the update is performed infrequently as it tends to be disturbing for other developers. Bringing about awareness in the sense that changes can be seen, but that they do not necessarily take effect at the same time, is difficult to achieve.

4.3 Long transactions

This model focuses more on configurations, logical changes and the fact that the development is done within a *group* of developers. The model

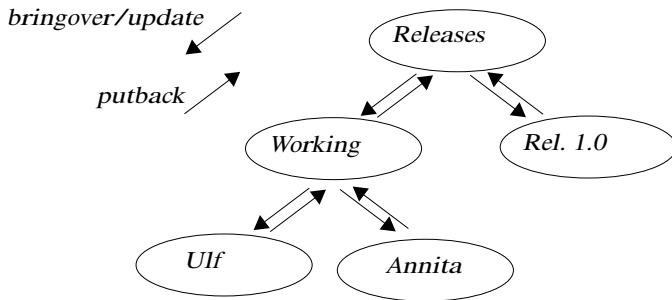


Figure 4 *Hierarchically organized workspaces direct the development process*

seizes upon the fact that the development of the entire system occurs via changes that are in bigger steps, by one or more logical changes, and by coordinating the integration of these changes.

The work process is formulated in such a way that a workspace is created and its content defined as an existing bound configuration. Changes are then done locally in that workspace. When the changes are finalized the result is used to update the original workspace as an associated operation¹. It is not until afterwards, that developers working in other workspaces can use the changes by first integrating them with the changes in their workspaces. The work in a workspace can be done by using a (local) version management. Versions of a file created in a workspace correspond to a temporary branch.

This model can be regarded as if working with configuration versions as well as file versions. A workspace corresponds to a configuration, a version of the entire sub-system. By using the operation *Bringover* one starts a new configuration variant. Changes made in files in that workspace also mean that the configuration is modified. Individual files can be version controlled according to the checkout/checkin model in the workspace, and without this affecting other workspaces. When the changes are finished (tested, inspected, approved etc.), they can be introduced as a *single* operation in the original workspace with the command: *putback*, which installs all changed files in the original space. It is only after such a put-back command that other developers can utilize the changes by performing the operation *update*, which merges the changes made in ones own workspace with changes made in the original workspace since the last bringover (or update). If files have been edited in both places, the update command means that the changes in these files are to be merged. Workspaces can be organized hierarchically which may be considered as a support for nested transactions, see Figure 4.

The model therefore discriminates between: (a) changes in an individual file, (b) publishing of a logical change and (c) the integration of published changes. The division between b and c means that there is support for a *conservative update strategy*. The possibility of managing changes in

1. The cycle of: copy, change and putback, corresponds to a long transaction in a data base context, hence the name.

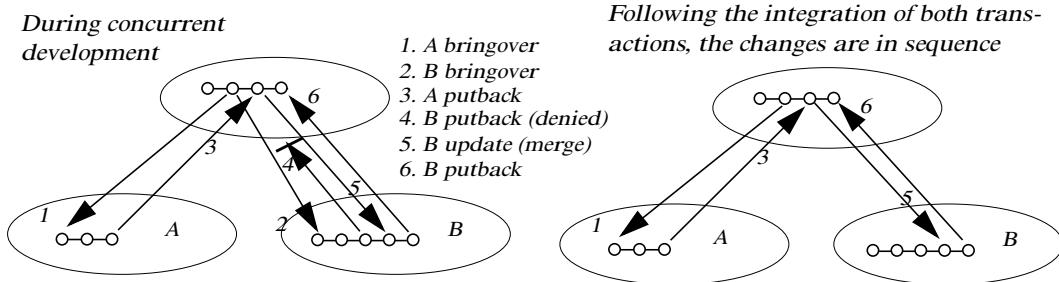


Figure 5 The relationship between transactions and versions within the workspaces, and the work process that is supported. Terminology from Teamware. The last transaction being completed in simultaneous work must integrate already completed transactions before it is finished.

several files simultaneously (b and c) means that the model manages *logic change* and tracking down to the modification of individual files. For instance the model can be used such that each change request results in a bringover/putback coupling.

Figure 5 shows how workspaces work as support for the individual developer and what happens during conflicts, i.e. overlapping transactions. In a workspace, i.e. during a long transaction, the checkout/checkin model works for individual files. There is no locking between the workspaces and many can begin transactions from the same workspace (1 and 2 in Figure 5). When the first transaction is completed, it is finished with a putback as usual (3). The detection of overlapping transactions is not done until the other transaction is completed (4). A conflict arises if the original space has been changed since the last bringover (or update) to the workspace (in this case caused by 3). Putback can then not be performed directly as the workspace (B) must be first updated with the changes (from A) and then changes from the two transactions are integrated in B. This is done by the command *update* (5) and if in addition, individual files have been changed in both workspaces these must be merged at this point. Following this a test of the result is best conducted in the local workspace before a new putback is attempted (6). The overall effect is, that what was for some time overlapping transactions, will afterwards be regarded as being arranged in sequence (first A then B in Figure 5) in the workspace.

The model is optimistic in the sense that it never prevents concurrent work. Parallel workspaces can always be created and the same files exchanged, which may lead to conflicts. Experience shows that in general, conflicts that are difficult to solve are actually rare in reality.

The model encourages an *optimistic update strategy* (i.e. to perform putback frequently) as the developer completing a last transaction is obliged to perform integration with his own and the already integrated changes. Developers in other workspaces can then be *made aware* of the changes in the original transaction but decide for themselves when to *use* them (conservative update strategy).

The model supports two levels of coordination between developers. Each developer/group can have its own workspace and the work in them can be done concurrently without them disturbing each other. Within a

workspace different strategies may be used. Each developer can have their own workspace, or several developers may work simultaneously in the same workspace. The situation between them will then be the same as in the checkout/checkin model.

Comments

This model works well in a distributed environment when a workspace has its own version control and therefore can 'manage on its own'. A workspace can be copied to another place, processed only there, and copied back without problems. The model was developed particularly for this kind of situation in distributed development with little or no on-line communication. However, in such a situation the overall view and awareness of what others are doing is greatly limited. However, there should not, at least in principle, be any problems in using the model in a system where there is automatic replication of workspaces. Accordingly, one should have the same possibility of seeing developments in other workspaces and to integrate with the original workspace etc., as in local development.

4.4 Change set

The Change-set model focuses on the management of logic changes, i.e. several associated changes. If, for instance, a change request (error correction, implementation of a new functionality, etc.) requires changes in several different files, this is called a *logic change*. A logic change can of course also include a sequence of changes in the same file. It is important to maintain the information that these individual changes are connected, as normally all of them should be included in the configuration for it to function as intended and for example, for the error to be corrected.

In this model, a system's versions are organized as a bound configuration, which is used as a starting basis, followed by a number of internally unrelated logic changes. Different configurations can then be merged by selecting which of the logic changes to include. In practice, not all logic changes can be freely combined, as there are often restrictions, such as that some cannot be used simultaneously and others that require each other etc. The original configuration, in combination with all of the logic changes can be used to create a large number of different configurations. If Change-sets are organized such that each change request results in a Change-set, a very clear connection between the request and the changes that the request actually resulted in, is obtained. A common example of where this model has been used is in the distribution of operating systems, which are often organized as a release with a large number of "patches". The patches that one then chooses to install depends on for example, the hardware one has.

The developer working by the Change-set model creates, often from starting out with a change request, a named logic change starting from a stable common version. The strategy for how and when such logic changes are integrated and tested can vary from that in which all logic changes are considered to be isolated changes, and are therefore not tested together, to that in which all logic changes are successively integrated and

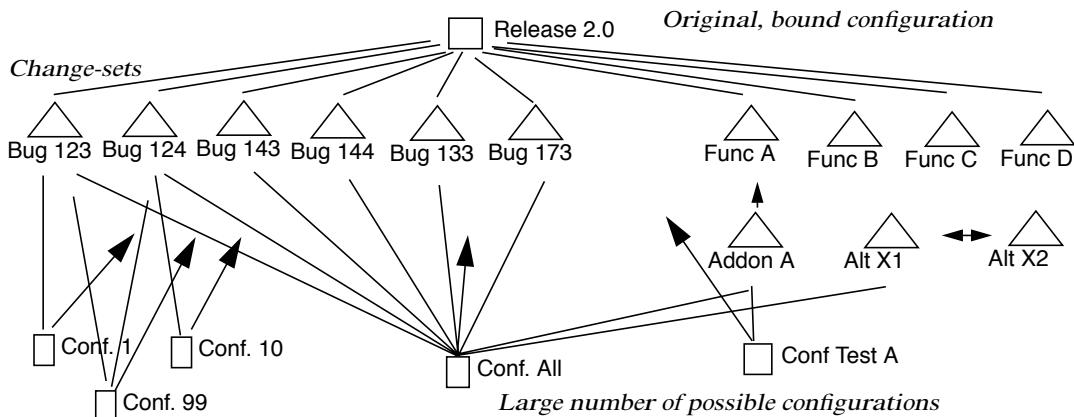


Figure 6 An example of a situation in which an original configuration and a great number of logic changes have resulted in Change-sets. These can be independent of each other, dependent (as Addon A is dependent on Func A), or excluding (as Alt X1 and Alt X2). The number of configurations of the different logic changes will rapidly become very large if they can be combined with complete freedom.

tested together in the product configurations (one or several) available. However, this means that in that configuration, the new logic changes are added as they arise. Conflicts, i.e. changes in the same file, do not have to be merged until a configuration where both are included is created, and possibly not at all. The Change-set model supports a conservative update strategy and almost encourages a conservative integration strategy - or at least it is not optimistic, in the sense that it forces the integration to occur.

The Change-set model has an obvious relationship with the long transactions model as a Change-set can be regarded as being the result of a long transaction. The difference is that Change-sets can be managed and named as units and their integration can be done at a later time point, possibly not at all, and sometimes even by the end user. In the previous models we have presented, the time sequence over which the changes are being performed, will, almost randomly, affect which of these possible configurations will actually arise. In the long transaction model, a sequencing of the transactions is forced as they must be integrated and tested one after the other, as seen previously. In the Change-set model, it can even be possible to test all of the possible combinations of logic changes and therefore all of the configurations that can be created.

In a distributed environment, the model can be used such that each location/group creates their own Change-sets. These can be replicated at different locations without technical problems. The model works well with a very conservative integration strategy, however, this may result in a weak support for group awareness at a local level, and even more so in a distributed environment.

Comments

The Change-set model has a number of obvious advantages for the management of systems with a great number of variants (such as operating

systems). The model gives a great flexibility regarding the creation of a suitable system from the component variants. If, on the other hand, the application is intended to result in *one* system and that all logic changes should be included, then perhaps the previous models are advantageous in that they force the early integration of changes, whereas in the change-set model, it can be made a policy. The obvious connection between change requests and the logic change can be a great advantage, perhaps in particular during the maintenance phase of a system. In a distributed environment, the model gives great freedom to the developers in one location to choose which of the changes made at another location, they want to use.

The disadvantages with this model are that several potential configurations may arise (all permutations of logic changes) and that it may be difficult to determine which of these are useful. Neither does the model support configuration versions *between* baselines, e.g. configurations intended to contain all of the error corrections. A work mode where such versions are created and then continually tested and updated therefore has no direct support, this may be particularly serious in a distributed environment.

4.5 Tool support for synchronization models

A CM tool can *handle* several models, but usually only one or two of them are really *supported*. In cases where there are two models, checkout/checkin is usually included as a part of a more complicated model. ClearCase for instance, can be said to manage checkout/checkin, but its views (or configuration specifications) makes it more reasonable to call the model Composition. In contrast, in RCS, one can simulate Long transactions by always creating a new branch at checkout. However, the model being supported in this instance is only checkout/checkin. Therefore, when selecting a CM tool it is important to check that the tool not only makes it possible but that it actually makes it easy and natural to work in the intended way.

4.6 Summary

Here we have presented four synchronization models. The borderlines between them are not crystal clear - partly because there are several aspects that are relevant which are not always independent, and partly because the models differ in several of these aspects. The models can be used more or less flexibly, in a number of various ways, such that the differences between them become even more diffuse. The models and how they are being used can often be understood and characterized in terms of the strategies that we describe in chapter 3.

- Checkout/checkin is focused on individual files, supports a conservative strategy for concurrent development, an optimistic update strategy and in practice, a conservative development strategy.

- The Composition model extends the checkout/checkin model with support for the connected version control of several files.
- Long transactions support the management of configurations of files, an optimistic strategy for concurrent development, a conservative update strategy and an optimistic development strategy.
- Change-set, models the changes rather than the versions and thereby differs from the other models. This supports an optimistic strategy for concurrent development, a conservative strategy for updating and a conservative development strategy.

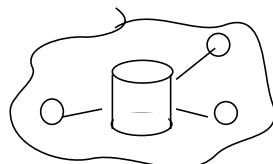
For a more detailed description of synchronization models we refer to [Fei91] and [Dar90].

5 Architectures

In chapter 2, "Distributed development", we discussed some different working situation cases where developers were distributed at different geographical *locations*. To meet the demands arising from these different situations, one can in a number of different ways, locate workstations and repositories/servers in these places, in different architectures. We will therefore from now on call a geographic place (e.g. the same house), equipped with repository/server and a number of work places, a "site". Developers with workstations but without a repository/server are therefore not a site. The different architectures being discussed are:

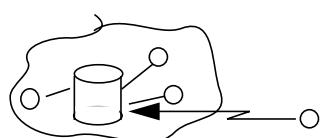
- Locally, a site at which everyone sits and works locally.
- Remote login. A site, but external developers are able to log in remotely, with e.g. rlogin, telnet or on www.
- Laptop computer to a server.
- Several sites by Master-Slave connections.
- Several sites with differing areas of responsibility.
- Several sites with equal servers

5.1 Locally to a server



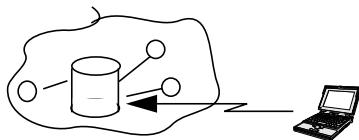
All developers are situated locally and work via a rapid network towards the same server.

5.2 Remote login



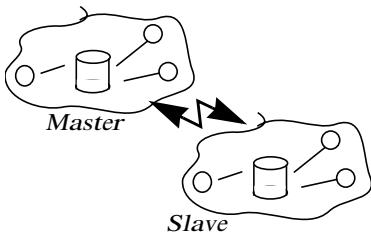
A single server towards which everyone works. Those situated at a different location than where the server is located, work towards the server by remote login, telnet, or other similar protocols. Technically a developer then works as if situated locally but is limited by a slower (and possibly a less reliable) connection, for instance over a modem or Internet.

5.3 Laptop computer to a server



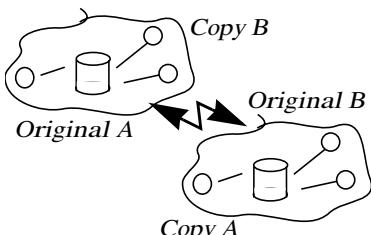
The server towards which everyone works, but in contrast to remote login, some of the product's files are copied to be then worked on locally. Updating and synchronization of the files is typically done on a daily basis. The work is performed without support by the CM tool. If a CM tool is available on the laptop computer, then a situation as outlined in chapters 5.4 or 5.5 may arise.

5.4 Several sites by Master-Slave connections



A version of a connected sub-system is copied from a master to another (slave) server where it is further developed. This architecture is commonly used without support from a common CM tool and therefore the version history is not included at the copying stage, nor at the following synchronization stage, at "delivery" back to the master. To avoid a complicated merge situation a sub-system copied to a slave server can not be changed at the master or any other server. If both servers have the same CM tool, and it supports this architecture, these limitations can be eliminated or at least reduced. Irrespective of this, the requirement for updating is usually relatively infrequent (weeks, months). A situation like this may occur with outsourcing for instance.

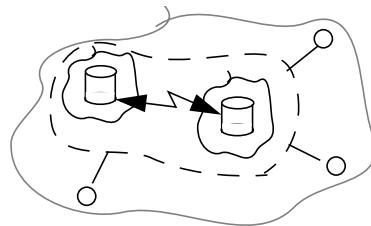
5.5 Several sites with differing areas of responsibility



Different sites are responsible for different sub-systems. The division can be based on the responsibility for certain files or certain variants. The variant concept must be the same for all of the files on the servers. For those parts that a site is not responsible for, the information can only be read. Synchronization is achieved by the changes in the original being transferred to the copy. Examples of such protocols are ftp, www, e-mail or those within the CM tool. Synchronization is often done automatically and at close intervals (time scale of hours) or when needed, i.e. when changes have been made. It should be noted that a site can have the original of one sub-system and at the same time have copies of others. This means that updating can occur in both directions between servers holding the originals for different sub-systems.

Compared to the master-slave architecture, this is a more permanent division and the synchronization is usually done automatically and therefore more frequently.

5.6 Several sites with equal servers



This is an architecture where several equal servers are located at different sites. These are automatically synchronized at close intervals (hours, minutes, seconds) and all of the servers have (with very little delay) the same information. The result is that a developer can work at any site (towards the server at that site)

without noticing a difference. The dotted line symbolizes how the servers at the various sites are being synchronized, the outer limit symbolizing a virtual site within which everyone works.

5.7 Discussion

The division in different architectures can serve as a guide when planning the introduction of distributed development, and as a basis for the analysis of the consequences and limitations of different solutions. The different examples may also serve as a guide for the manual management of the situation and to understand what different CM systems can offer. Of course, it will never be very easy to use an architecture until there is tool support.

The situation of remote login is often used in manual management. Situations in which temporary branches need to be merged will most likely increase, therefore a tool supporting this function would be of great value. The same principle applies to the situation with simple use of a laptop computer. There are also simple tools for supporting the synchronization of directories with files existing on the server as well as on the laptop computer, e.g. File Assistant for Macintosh. The Master-Slave connection is a situation that is directly supported by the tool Teamware, which was developed to support this particular architecture. Similarly ClearCase Multi-Site supports the architecture in which there are sites with differing areas of responsibility. Finally we conclude that the ideal situation with equal servers, is unfortunately not covered by any of the commercially available systems.

The interval between updates will impact on when changes can take effect and thus controls the degree of awareness in and between the groups.

5.8 Summary

There is a strong connection between the different work situations described in chapter 2, "Distributed development", and the architectures described in chapter 5, "Architectures". Of course this is not really a coincidence, but different needs have induced the development of different architectures and conversely different working situations have been used when new techniques become available. The simplest cases of working situations, locally and distance working, are situations that probably always exist and which must obviously be supported by every architecture solution. Outsourcing and co-located groups at different locations are situations where either one or both of these may apply. Here it may be of interest to select the architecture depending on which case is considered to be most important.

Finally, the situation with distributed groups is problematic as, in practice, it requires the most advanced architecture, i.e., equal servers, which is not yet supported by existing CM tools. Thus in finding oneself in this situation, one is forced to manage it by partially manual methods.

6 Experiences and advice on key areas

When software is to be developed in a distributed manner, the working mode must be reviewed. The work mode gives the basis for which tool support is required, perhaps tools must be supplemented or exchanged. This need for reviewing also applies to companies with a well-established local CM. In this chapter, a number of areas are indicated where a distributed development affects the working mode and gives rise to situations which must be solved in order to obtain an effective and problem free development. The areas being discussed are:

- Concurrent development and awareness
- Change management
- Incremental development
- Security
- Data storage, repository
- Integration and delivery
- Consistent development environments

For each area we give our definition of the area and examples of problems that may arise or be magnified by geographical separation, we report on the experiences from the interviewed companies, as well as advice and practical guidance.

Different companies have different problems and the same solution cannot always be applied to similar problems due to the appearance of the product, security demands and other issues. By describing the situation for several different companies and the way they handle their own situation, we hope to supply experiences that could be useful for many. Each area is completed with a practical guide, where the focus is on the work model. The advice contained within is relatively specific however it should be read in such a way that it is adjusted according to the reality of the situation in which it is to be implemented. The tools as well as the practical advice contained within the introduction are discussed in separate chapters at a later time point.

6.1 Concurrent development and awareness

To shorten the development time, companies are increasingly developing more of their software in a parallel manner, i.e. a number of developers make simultaneous changes to the same product. A prerequisite for this is that the developers can and may make simultaneous changes. By letting developers work in different parallel development branches they can, despite the concurrent nature of the work, work in an isolated ‘sandbox’ and in that way avoid using each others temporary changes. However, the isolation results in the possibility that, without being aware of it, they may make changes that are in conflict with each other. These conflicts have to be solved when the branches are subsequently merged into a common development branch.

Concurrent development can be achieved at different levels. At the system level, sub-products or different functions are developed in parallel (concurrent system development), and at a lower, more detailed level, several developers can make simultaneous changes in different versions of the same file (concurrent software development). The lower the level, the greater the possibility for a high degree of concurrent work, but the greater the risk of conflicting changes.

There are two main ways of reducing the risk of conflicts:

- A good product structure makes it possible to make dependencies obvious and distribute the work to different parts of the product. The more independent the parts are of each other, and the better their interfaces are described, the smaller the risk of conflict. It is then relatively easy to distribute the areas of responsibility, as different parts of the product, to the different developers or project groups.
- A high degree of awareness of what other developers/project groups have worked with and are currently working on. Concurrent development on a low, detailed level requires more awareness than on a higher level. Working concurrently at the system level, knowing if/when an interface is being changed may be enough, whereas two developers working in the same module would probably like to know if/when the other starts working in the same file.

Awareness is mainly achieved through formal and informal meetings, e-mails between developers and through CM tools. An implemented process can for instance send e-mails under certain predefined conditions, one can see who created new versions, when and so on. However, in a distributed situation much of the information obtained through the informal meeting disappears, in addition, the formal meetings become less frequent and may be performed as telephone and video conferences rather than sitting around a table. Other aspects, such as cultural differences and time zones, also reduce the possibility of good communication. To increase the degree of awareness in a distributed situation it is therefore required that this reduced communication is compensated for by an adjusted work model and CM tool support.

Well defined tasks result in increased awareness, in the sense that a person knows what the others should be working with; this is an important aspect with collaborations over long distances. However, it gives no "real" awareness through system support, i.e. what actually happened yesterday, or what is happening right now. By defining the tasks for each developer one does not decrease the reasons for concurrent work at lower levels either.

The points above should, however, not be taken too far. When the division of responsibility is too great, it easily becomes too static and therefore limits the developers in what they can do. Associated tasks, e.g. change requests, are broken down into too small pieces which should be performed by different people in their respective areas of responsibility. It is better instead to enable a developer who notices a simple error in another persons module to, at least temporarily (possibly in an own

branch), quickly correct the error in order to be able to test his own changes. This is particularly important when the developers are situated in different locations (although it may be more difficult to achieve). The person responsible for the module should then be informed that somebody else has made a proposal for a change, to be able to decide whether it should be integrated (merged) into the main branch. There is also a risk of the projects becoming more sensitive to staff changes and illnesses when only a few, or in the worst case only one person, understands a particular piece of code.

The support for increased awareness is sensitive; for instance, too large a number of messages intended to increase awareness can hinder more than they help. That may be the case if small changes far down in a sub-project, result in messages being given to developers working higher up in the product structure.

To prohibit concurrent development at the file level for example, means that two developers will not be able to simultaneously make changes in the same file. In contrast, it is possible for the two to organize the work in sequence and implement their changes one after the other, i.e. there is dynamic locking of the files. Areas of responsibility result in a static division where (always) only one of the developers is allowed to make changes in a particular file. The strict allocation of exclusive areas of responsibility is a firmer synchronization for the developers than just prohibiting concurrent development at the corresponding level.

The following questions must be answered:

- At which level can concurrent development be managed in distributed development?
- How much awareness is required for the selected level of concurrent development and how is it achieved, within projects at the same location as well as between them?
- How should interfaces between the modules be managed? Whichever way the work is divided, the interface descriptions get special treatment.

Experience of concurrent development

New development and maintenance have slightly differing characteristics, which makes it possible to handle them differently.

With *new development* it is often possible, by having a suitable product structure, to split the work between developers such that they become responsible for different parts of a product. Clear tasks and well-defined interfaces result in the work, particularly between locations, being done quite independently. Despite this, awareness is important and companies are now looking for a greater connection between development at different locations, with a possibility of seeing on a daily basis how work is progressing, rather than as now with a regular follow up of, for example, once a week. Since only one person is allowed to make changes in a file due to the division of responsibility, branches at the file level are rarely used.

To make the division possible, it is important that the description of the interfaces are defined, scrutinized and approved early on, and that they quickly come under strict change management.

During *maintenance* it is more difficult to make the same division. A change request affecting several areas of responsibility may be difficult to work with if, for each file/module, one has to contact the persons responsible and request that a change is performed. As an alternative, someone responsible for the entire change request can create a temporary branch for the entire product, in which the whole change is implemented, the branch can then be merged back to the original branch. The actual merge can be done by those responsible for each file/module.

Figure 7 shows the two different ways of dividing the responsibility between developers.

However, some people believe that the working mode according to “new development” outlined above is only caused by an inadequate support for distributed development and that it is a concession in order to get simpler CM management in the absence of branches. Instead of this, the normal working mode should be that every new functionality is implemented by a responsible person, who then implements the complete change in all affected files.

It should also be stated that the maintenance of, and further development of a product are the greatest part of its life cycle (at least 80% according to some sources). Therefore the formulation of tools and processes just for new development rather than for further development/maintenance is a common, and in this respect, serious mistake.

The working mode also differs depending on the type of code used. For very special codes, for example difficult control algorithms, it is more common that the individual responsible for the code makes the changes for himself.

Some companies use a pure checkout/checkin model with locking at the file level and do not allow the creation of branches, even at maintenance. As several files often have to be changed to implement a complete

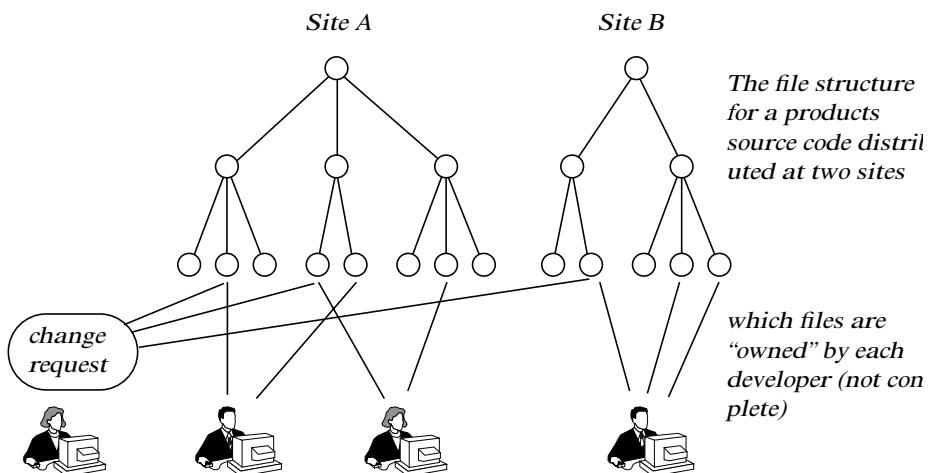


Figure 7 *The source code files of a product distributed at two sites and with two ways of dividing the responsibility: (1) the developers are responsible for their files/modules or (2) the developers are responsible for their change requests/functionality*

change request, a “deadlock” between two developers may arise when they both want to change files which the other one has already locked. If one has locking without branches, a method to solve all arising deadlocks must be created. The most common method in local development is that the developer who has got the longest time before his deadline (manually) reverses his changes, after which the other developer can complete his change request. In some way (without direct tool support) the waiting developer becomes aware that he can continue, the previous changes are repeated and the change request is completed.

In many companies there is a fear of working concurrently at a lower level (module or file). It is unclear whether this fear is due to the fact that they want to keep the positive aspect of clear areas of responsibility (“only the one understanding the code can make the changes”), or whether it is due to a fear of loosing control of what is happening in the project and who does what. These fears are to a large part due to poor tool support for the maintenance of awareness.

One conclusion that can be made is that two sub-products (in a well-structured product with normal dependencies) can be developed more quickly and easily if they are both developed at the same location rather than at different locations. However, it is still an open question as to exactly what support is required by the CM tool to avoid this difference or even whether it is possible to avoid it. In addition to awareness support, which was discussed previously, many tools lack an overall picture of how the system as a whole is developed. For each individual file one can see which versions are included as well as the differences between them. In contrast, configurations (the product and the sub-product) are described in a configuration file and usually lack an own version graph where one easily can see how they have developed. In addition, a version graph would clearly show if, for instance, a certain sub-product was at that moment being developed in parallel by several developers.

Practical guidance

- Replicate the information at the different development locations so that everyone can work towards a local server. Synchronize the replicates regularly and frequently, preferably automatically using tool support.
- Do not lock files to prevent simultaneous development, particularly if the lock restrains developers at other locations. Instead, make it possible for the developers to create a temporary branch themselves. The branch should then be merged with its original branch as quickly as possible and finished.
- Create a good product structure that gives an early, natural division of the work. A good structure decreases the need for branches. Where branches are still being used, a good structure reduces the risk of conflict at the subsequent merge.
- Do not use too strict a division with people being responsible for individual files or modules. This easily leads to static and inflexible change management where several change requests affecting the same files, cannot be managed concurrently, particularly if the

same change request may affect areas of responsibility in several places.

6.2 Change management

The management from error report to actual changes must work even when the error reports as well as the actual changes are being made in a distributed organization. Figure 8 shows an example of an architecture for the reception, storage and division of a change request. The local support, for example in the form of the sales organization, receives error reports and put them in a central database. The variant with only one central support also exists, but a local department usually has a better level of contact with the customers, knows their installations better and can directly manage human errors, incorrect installations and such like. A central decision forum (e.g. CCB - Change Control Board) processes the different requests (e.g. by criticality and urgency) and allocates them to the correct development location. The actual allocation can be implemented by anything from letting the request stay in the central database and just changing the status of an attribute, to physically sending the request to a local database. The development locations also deposit error reports in the central database, for instance when errors are discovered during a test.

What differs in the architecture described compared to local development, is that the development locations are geographically separated. This may lead to:

- it becoming more difficult for (the central) decision authorities to meet as they consist of representatives from the development loca-

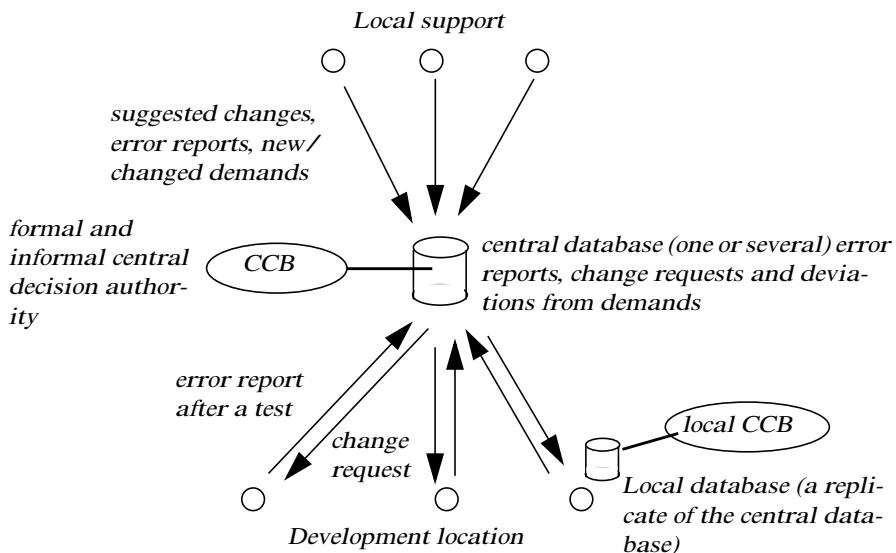


Figure 8 *The role of the database in change management*

tions for example. If they meet infrequently, it may result in a slower turn-around time for the projects and a slower feedback from changes that have been done. To increase the meeting frequency, telephone conferences rather than real meetings are arranged. However, telephone conferences are not as effective and the risk of misunderstanding is greater.

- a need for local decision authorities as a supplement to the central ones to compensate for the point made above.
- that local databases are created. To increase the availability, replications of the central database are made.

Maintaining error corrections to a released product can be done in the products own maintenance branch. It is then important to also make those changes in the main development branch and possibly in other released products containing the same error. Additional products have to be checked to see whether a correction would be relevant. Also, with distributed development, when products are now owned and located at different sites, collaboration on the reuse of the actual error reports as well as the use of the final error corrections is important.

It is also important that each change refers back to (or starts out from) a bound configuration. Following the introduction of the change, a reference as to which configuration the change has been introduced should be included.

The management of changes is described in the CM plan and should follow a procedure defined therein. One such example of a procedure is illustrated in Figure 9, where six steps are defined:

1. Create and register CR (Change Request).
2. Analyze the contents and influence of the CR. Basis for CCB decision.
3. CCB decides whether or not the CR should be implemented. If there is any uncertainty the CR can send the request back for further analysis.
4. CR is implemented.
5. CCB verifies that the change is correct, i.e. that the test results or some other form of evidence shows that the change is implemented

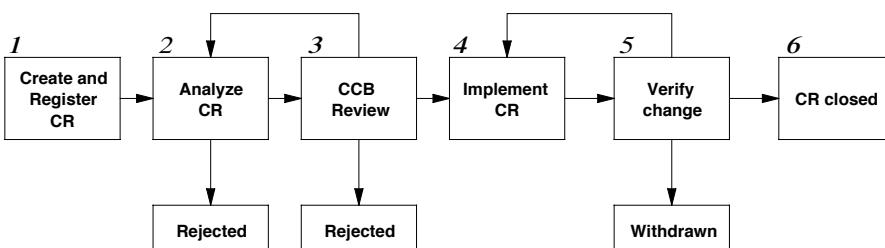


Figure 9 An example of a Change Request procedure

and works. In some cases the change may “no longer be relevant”, in such a case the CCB can withdraw the change.

6. If everything is OK the CCB can close the CR.

This is but a brief description and more details could be added. With distributed development, it is possible that there are several decision forums with differing areas of responsibility and perhaps also at different locations (see above). The CM plan must clearly describe how these forums should cooperate and how much synchronization there should be between the local decision making authorities.

As we have previously concluded, the awareness of what is happening at other sites is reduced with distributed development and there is a risk that an individual does not feel as loyal to the products derived from other places as to their own. In other words, it is important to give as much support as possible to following the correct procedure, both in the form of a clear procedure outlined in the CM plan as well as in the form of tool support. This should be through awareness and strong support for the merger of different product branches whilst maintaining the ability to track what has already been done.

Examples of questions are:

- How is an incoming error report or change requirement managed and who decides where to direct it (at which project, at which site)?
- How are error reports, change requests, and so on stored? In one or several databases, centrally or relocated?
- How are changes in the databases managed and how does one assure that potential copies (replicates) are updated?

The directing of error reports and change requirements is also an economic matter. How does one manage an incorrectly placed error report? It takes time for the person receiving the error report to notice that the report has been directed to the wrong place, the question is who should pay for that time?

Experiences of change management

Change requests can be managed by both formal and informal decision forums. One example of a formal forum is the CCB. Examples of alternative names for error reports include: incident report, defect, trouble report, and anomaly.

It is common that a central database is used for error reports and many people stress the importance of a common error management system. However, distributed local databases for error reports also exist. The advantage of local databases is that they are easily accessible to both developers and project management. However, in the cases where the same request is stored centrally as well as locally there is a risk that they may be inconsistent. The risk is particularly great if it is unclear as to who is allowed to make changes in these databases. Automatically synchronized databases are rarely used.

Another risk is that the people responsible for the project keep their own error lists separate from the common database. Such lists tend not to

be made public for other products, which of course impairs the possibility of cooperation and reuse. The best way of avoiding this is (again) by having a well-defined working mode and tools supporting the process.

An alternative to a central database is to divide the database into several parts, each managing different types of reports. For instance ABB Automation Products has four levels of databases: (1) "CCRP - Customer Complaint Report" which stores complaints from external customers via the sales organization, (2) "PMR - Product Maintenance Report" which manages error reports in released products, (3) "PPR - Pre-release Problem Report" which stores error reports in beta releases and (4) "CR - Change Request" which stores change requests.

There are several reasons for such a division, one being that the different databases can be implemented using different tools adapted to each developer, e.g. CCRP in LotusNotes for customers and sellers, and CR with RCS for the developers. They may also have different distribution. CCRP and PMR are global whereas PPR and CR are internal.

However, this results in a certain degree of overhead and it must be possible to fully track the movement of a message between the different databases.

Change management must follow strict routines. Errors discovered in a product will result in change requests in one or several components, at one or several places. It is also important to have clear rules as to who closes/finishes a change request when the error has been corrected. This is particularly important if the changes are performed in different locations, which makes it more difficult to control whether things that should have been done, have actually been done. It must be possible to follow up these matters and the status of each change must be made public. For instance, ABB Robotics Products have a www interface for their Oracle database to increase their accessibility via the intranet.

Saab Gripen have introduced support into their system to keep changes together belonging to the same change request. If a change request affects several modules, which therefore have to be modified (new versions created), a new version of a module cannot be used in a particular configuration until all of the modules are finished. In that way, the use of inconsistent module configurations is prevented.

In large projects, hierarchical CCB's may be necessary, i.e. central as well as local ones both being responsible for their own developed product. Only matters affecting several CCB responsibility areas are managed in the central CCB. This reduces the burden on the central CCB, which may find it difficult to have as close a meeting as otherwise required.

Practical guidance

- Regularly check all opened, ongoing change requests and their status.
- The CM plan should describe the change management process. With distributed development it is particularly important to carefully describe how changes are finished (closed).

- In cases when relocated projects have their own CM plans, it is good to have a generic CM plan to which the project's CM plans refer, indicating differences from the generic CM plan.
- Avoid using several different databases for the same kind of change requests. For example, the product management and the software development should not have separate databases for error reports, and the project leaders should not have their own lists separate from the database.
- Let each change refer to the bound configuration in which the error was discovered as well as the configuration in which the change is implemented.

6.3 Incremental development

As an alternative to the more traditional 'waterfall' model, incremental development may be used. Incremental development is a way of organizing the project work such that integration and testable results are achieved early on. The product is developed gradually, in small, well-defined, steps. Each step, or increment, results in a functional and evaluated product.

The model is based on a prototype-based development. A prototype that may only consist of an empty shell, but which is still an integrated product, is created early on. Each increment adds functionality to the prototype, consequently this is continuously integrated and tested, until finally, all of the product's functions are included. Provided that a simplified development process, consisting of the three-step sequence: analysis, implementation and test is being used for each increment, the total development process can be graphically visualized according to the example in Figure 10. The waterfall model is shown above the time axis. Overlap between the three phases is possible, but it is always by iteration. Below the axis we show how the development is divided into several increments following a common preparatory analysis. All of the three phases of an

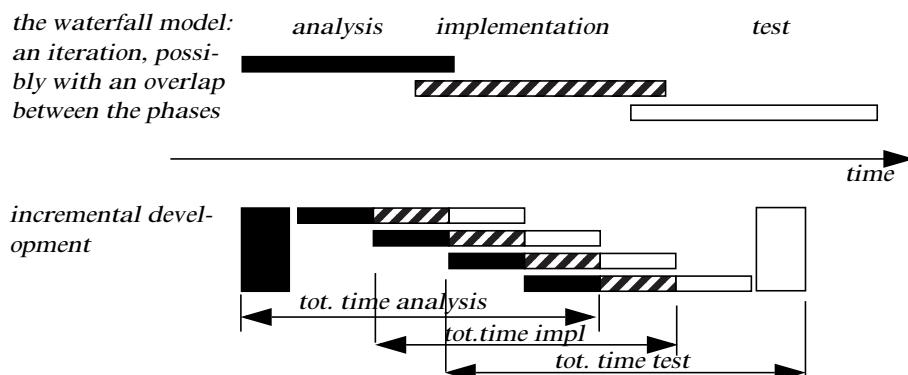
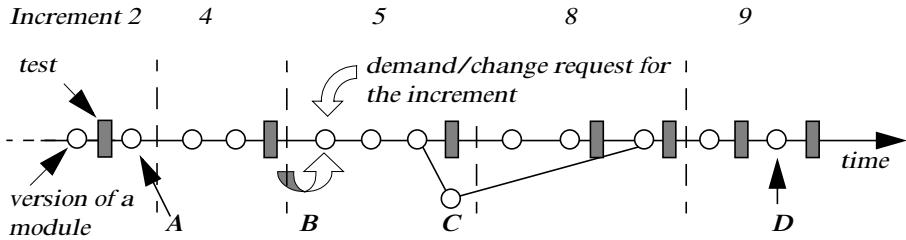


Figure 10 Example of a development process with incremental development. Despite the fact that the total development time is shorter with incremental development, the total time of the phases is longer.



- A) new version with a bug-fix to an error found in an increment test
- B) errors found on test of increment 4 lead to a change request in increment 5
- C) error implemented by increment 5 is found at test of increment 8. Corrected with a bug-fix branch.
- D) error implemented by increment 8 is found at test of increment 9. Corrected directly in increment 9.9.

Figure 11 A module's version history by the increments and different ways of managing errors found at test of an increment.

increment can, to a great degree, be performed concurrently with the development of other increments. Thus the total time for each phase may be longer in the corresponding projects using the waterfall model, although the total project time is actually shorter.

Incremental development results in an optimistic development strategy. Each increment is integrated with the one built previously and is tested directly, this gives an early indication as to how well the developed parts work together. In that way, the work developing the subsequent increments can make use of the results from previous evaluations, and not continue using imperfect solutions. At best, new demands from the customer can also be taken care of in increments that have not yet been started. In addition, functions implemented in early increments are tested several times in later increments, this reduces the number of errors close to delivery.

In this context, the waterfall model is comparable to a large increment where the entire product is to be included in the first test, i.e. according to a restrictive development strategy with “big bang” integration. There is a great risk of extensive revision being required, which may lead to delays in the project.

As usual, the product can be functionally and physically divided into several sub-products/modules. If a model is developed in several increments, a development through the increments according to Figure 11 is obtained (only the increments in which the module is modified are shown). In the example shown, there is no concurrent development of the increments affecting the module, which had required division into branches and subsequent mergers. The Figure also shows four different ways (A-D) of correcting errors discovered at an increment test.

A shows what is the perhaps most common case in which the error is directly corrected by the increment. In B, an error has been found, which is corrected in the following increment instead. The change request that the error leads to, is taken care of at the same time as those already planned for the increment. Errors can also be found in the functionality

implemented in a previous increment (i.e. was missed during the tests of that increment). C shows the situation where the responsible increment has the error corrected, this is performed in a branch specifically created for this purpose. The bug fix is then merged into the main branch where the increment is tested again. The error may also, as in D, be corrected by the increment that found the error.

Incremental development makes greater demands on CM in general. There are more parts to take care of, more versions and a greater number of baselines, which of course increases the complexity.

With distributed development, it is natural to try to keep the increment groups together, i.e. those working with the same increment being situated at the same location (otherwise the situation described in chapter “Distributed groups” is obtained). Therefore, the occasions when communication between the different increments is required, are important. Examples of such occasions are:

- Integration and test occasions always require communication. These increase in number with incremental development. However, they do not become very large and difficult to manage.
- In the four cases of error correction visualized in Figure 11, communication between those responsible for the increments involved (the one being tested and the one in which the incorrect functionality was implemented) is required along with the CCB, who decide which of the four bug fix methods should be used.
- The CCB distribute demands to the person responsible for the different increments at the different sites.
- A module being developed in several increments (at different locations) must be kept consistent. It is best to have somebody responsible for the module (*design item coordinator*) who becomes the link between those responsible for the increments. This is a role, which must be managed over several sites.

Experiences of incremental development

To be able to view the whole at the same time as the increments are being developed independently, perhaps at different locations, a cooperation between the different people responsible (roles) for the incremental development is required. Those responsible for the individual increments manage demands and change requests for the increment, whereas the person responsible for the module (*design item coordinator*) approves of all enlargements to be done in the module he is responsible for. In this way, each module can be kept consistent despite it being changed over several increments. Those responsible for the function ensure the interplay between all modules.

Figure 12 shows an example of an organization for incremental development. Each line supplies several projects with resources/competence. The planning of the work in a sub-project is based on the functionality that should be added to the system in the various increments.

- *Line Manager (LM)*: Is responsible for *who* shall do the work and *how* it shall be done.

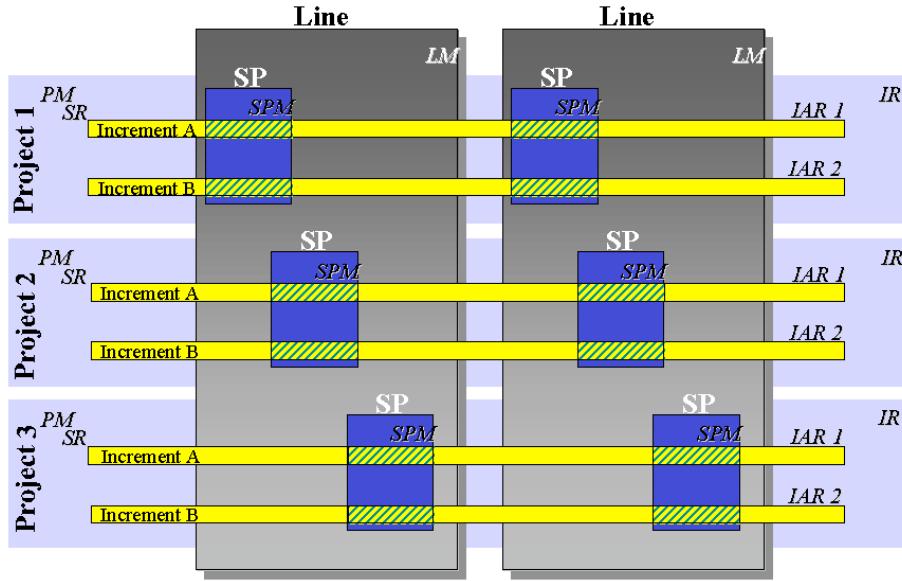


Figure 12 Roles and projects at incremental development

- *Project Manager (PM), Sub-project Manager (SPM)*: Is responsible for what shall be done, and when it shall be done.
- *Increment assignment responsible person (IAR)*: Is responsible for the accomplishment of the increment assignment.
- *System responsible person (SR)*: Has the technical responsibility in the project.
- *Integration responsible person (IR)*: Is responsible for the increment plan.

The risk that different sub-projects diverge during their development is increased when the developers are geographically dispersed. If they diverge, a restrictive development strategy will result in a complicated “big bang integration”. Incremental development avoids this by the early use of integration tests.

Practical guidance

- Both the whole, in the form of functionality, as well as each increment should have a person responsible and it is important to make the interplay between the responsible persons work
- Introduce a strict and early CM. For instance, incremental development results in a greater number of baselines than does the traditional waterfall model, this increases the complexity.
- Change management is particularly important. It must be known what has been done in one increment and what will be done in the next.
- Use a name convention such that it is clear as to which increment a baseline belongs.

6.4 Security

By *security* we mean the need to being able to limit access to certain data. To achieve a more dynamic organization, roles rather than people are given the different privileges. One person can play several roles and one role can be held by several people. The privileges are changed as the project passes through different phases. For instance, it may be advantageous if the developers are no longer able to make changes to a code once it has entered the test phase. To further control these privileges, they should only be applied to the files or parts of a product that affect that particular role. Figure 13 shows an example of how privileges can be placed and visualized. Figure 13a shows how the privileges for different roles can be defined for each stage of the development process. In Figure 13b the product model is shown in the form of a tree structure, where the privileges for separate nodes or whole sub-trees in the model are given different roles. The example in the Figure indicates that the role developer has privileges for sanctioned work in the development process, and then only in the areas of the product indicated by the dotted lines in the figure.

Roles and access rights are often used as a part of the implementation in the development process, i.e. also when no security demands are made. However, when security is required, it is important that access protection is implemented at all levels. From physical scale protection, to storage media, operating systems and to roles with certain privileges.

By *protection* we mean the use of actions to prevent or complicate access by unauthorized people to access protected data. People who, according to the established access rules, are not authorized will be prevented from gaining access. Protection also includes data integrity, which means that it should be impossible to falsify data.

Protection and security demands often conflict with other demands such as the reuse of code and resources. In particularly in distributed development, many of the solutions are based on the fact that information is being sent between the sites. Open networks that for example, enable awareness, may then be in conflict with information privileges.

In cases where secret defense, or other very sensitive information is managed, a demand for two physically separate networks may arise, one for secret information and one for all other information. A separate net for the secret information facilitates the implementation of access protection. The disadvantage is that the whole system can not be stored all together,

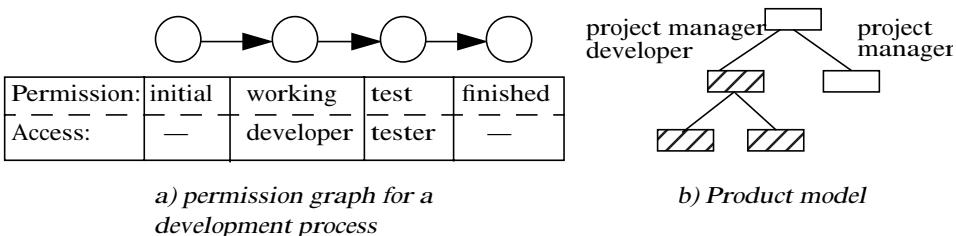


Figure 13 *Access table for the role "developer"*

which makes working more difficult for those who need access to both secret and non-secret data. Two networks also means that the two separate development environments should be kept alike, see chapter 6.7, "Consistent development environments".

As an alternative to two networks, a single secret network, in which all information is stored, may be used. Software usually has to be managed in this way as it is impossible to compile software in two networks without there being a connection. The disadvantage is that all management must be performed according to the protection classification of the most secret information.

It can also occur that metadata for secret documents may be managed as open data. In that way a complete picture can be obtained in the open network which facilitates document management and configuration management. However, as it may be difficult, for protection reasons, to automatically keep the actual documents and their metadata consistent, there is a risk of divergence.

For certain kinds of secret data, it is sufficient to write the actual document in cipher, the documents can then be stored mixed with open ones in the same open network, this considerably reduces the problems.

Examples of questions are:

- May we distribute everything between the different sites?
- Can everything be replicated? May all clients/servers access metadata as well as the actual files? If so, do we know the privileges for each client/server and are they changed as time goes on?

Experiences

Companies handling defense secret information, with, for example FMV as a customer, have very special security requirements. Requirements such as physically separate networks between different projects, no "export of war material" in the form of requirement specifications, source code and so on and "data transfer" via diplomatic post, result in the fact that many otherwise suitable solutions cannot be used. The companies tailor-make their solutions in collaboration with the customer and it is difficult to find any general experiences that companies without these requirements can/want to use.

Security is not, at least in the first instance, directed towards the employees but to the uninitiated and to a certain extent towards errors and mistakes. In contrast, as a part of the process implementation separate privileges may exist between projects and access rights may prevent the modification of certain files. One way of safeguarding the employees is to not give them any more information than is necessary. In this way, the risk of them receiving unpleasant inquiries regarding secret information is reduced.

Protection is usually solved either by hired lines or by writing in cipher, both of which by themselves are considered to be safe. However, writing in cipher can not always be used in international projects. For instance, France does not permit the use of cipher.

More or less all companies protect their own internal networks with firewalls. Most have their own modem pool inside the firewall and some kind of secure call-up, e.g. a password generator and/or dial back.

It is important to have the right protection and security levels to achieve a good level of *actual security*. With too high a level, there is a risk that “short cuts” are used, resulting in a lower degree of actual security than if the requirements were lower in the first place and were actually being followed. A too high security level is also more expensive and results in a longer development time.

Practical guidance

- All information should have *information owner* who is responsible for the life cycle of the information, that it is correctly classified, distributed to the right people and that the approved systems are managing the information.
- Implement a security level that is sufficient and does not bring about unnecessary extra work for those using the system.
- Implement security at all levels, from a physical protection and storage format, to roles and access rights.

6.5 Data storage, repository

Storing all data in a safe way is of course important, both during the development phase as well as after release. During development, one does not wish to risk the loss of work already completed, in addition one wants to keep the possibility of backtracking in a project if it turns out that one has made an incorrect decision. Therefore, all versions of all the information are stored in a *project repository*, usually implemented by the CM tool.

To be able do something about potential error reports after release one must be able to recreate the product as it was before the release. This not only includes the source code but the entire environment must be recreated for the error to be verified and a new version of the product to be built. Therefore, all released products are stored in a *product repository*. This can be implemented by either putting a “tag” on all files included in the product or by copying the file structure of the bound configuration and storing it separately, e.g. on tape or CD¹. A product repository contains all that is needed to recreate a product. In addition to the source code, all of the delivered generated files (binaries) as well as documentation are also stored. No matter how the product repository is implemented, it is important that somebody is responsible in the long-term for the released products.

1. There are two things that one protects oneself from by copying the product to another format on tape compared to keeping it within the CM tool, these are: (1) mere errors in the CM tool which result in one suddenly being unable to access data or that data is destroyed and (2) that the storage format used by the tool becomes out of date and can no longer be read with the functioning equipment.

In a distributed situation, one has to decide whether the project and product repositories should be central or whether each site should file "their part". As each site has its own original archives, these and their contents may be linked to a common concern register. The register can, in principle, consist of pointers to the sites where the information is stored locally.

Due to security requirements there may also be a need for all sites to have the entire repository stored locally. If so, it is important that these are synchronized.

Experiences

During the development phase, most people use a tool offering configuration and version control. Within these, there is the possibility of freezing versions for individual files, but also of freezing entire configurations (of frozen files). In combination with backup, this gives a satisfying level of safety during the development. Each site manages its own backup. If everything is replicated, this of course means that backup is performed on all data at each site, this being a duplication of work which is not considered to be a big problem.

To further increase the safety for released products, certain product repositories, which are slightly different from the project repositories, both technically as well as administratively, are often used. The project repository contains everything from frozen configurations to files being changed. In contrast, a product repository only contains frozen files and documents and is therefore more inflexible.

If one has really frozen a configuration in a project repository, this should of course be impossible to change or remove. Given this, many companies still choose to "copy out" everything that is included in the released product and store it on (one or several) tapes or burn it onto CD. In addition, the hardware being used during the development of released products is also stored, as it may be difficult to obtain at a later time point. Everything is stored in order to be sure of being able to recreate the situation exactly as it was at release.

Administratively, responsibility is given to a product administrator as a released product comes into the product repository. This is done irrespective of whether any physical copying occurs or not. Project managers, who are normally responsible for the project repository, will soon have new tasks, whereas a product administrator can guarantee a long-term undertaking.

Practical guidance

- Use the CM tool both as a project and a product repository. At external release, the entire development environment should also be stored in the product repository.
- With long maintenance times or when different CM tools are being used at the different sites, a copy of released products should also be stored in a tool independent format and media.

- When the data storage issue is being checked, a possible security and protection classification as well as obligations must, of course, also be considered.

6.6 Integration and delivery

With distributed development, the responsibility is often distributed to the different sites with integration occurring just before release. This results in a relatively simple CM management during the actual development phase. Each site manages its own version control and only the changes in demands and interfaces have to be managed for the whole product. In contrast, the product integration is complicated. There are two different integration strategies:

- Central integration. All sites deliver to a person responsible for the product, where everything is integrated and built on a central basis. Testing is performed until either a number of errors are found and the different sites have their parts returned with error reports, or, until the product is released. Alternatively, only a central test and bug fix is done until there is a rudimentary basic functionality. Then the whole product is sent to all sites which themselves test and fix their parts. The whole process is iterated until release.
- Local integration. The sites deliver to common libraries after which each site itself integrates, tests and corrects errors. The process is iterated until all sites are satisfied. The deliveries do not have to be synchronized between the sites. When a site has corrected its errors, it can directly deliver the new version. A conservative update strategy should be used (i.e. other sites decide for themselves when a new version should be used) such that test series should not be destroyed at the other sites. Awareness is important to ensure that a site is aware of when new versions are to be delivered and the test environment can be updated.

The whole procedure at delivery is similar to that for outsourcing and for ordinary delivery to a customer. This means that delivery occurs more often with distributed development than local development, this is why the delivery process must be settled early in the CM plan. Also, the configuration revision process described in the CM plan is important for deliveries between sites. It is important to know exactly what is included in a delivery and what is different from the previous one.

As integration and testing is a rather laborious process, it can be a good idea to reduce the number of released versions of a product. As an alternative to building more variants for customers with differing demands, only one version of the product, which includes all of the functionality is built. Instead customer specific functionality is obtained by the use of certain configuration files, which are delivered with the actual application. Such a file does not have to be very complex and can be man-

aged by *one* site, this gives a complete picture of what the customer receives.

Some other questions which have to be solved are:

- How does one obtain information about all components included in products that are at different “sites”?
- How is the integration of different components synchronized?

Experiences of integration and delivery

In the cases when the customer obtains a source code, an executable code should not be delivered, but only the source code together with detailed instructions on how the product should be built (beginning with the installation of the operating system). The reason for delivery of a source code is usually because that the customer himself wants to maintain the product at a later point. By building everything now, they ensure that they can really follow the building instructions.

Delivery of sub-products developed at different locations is done to the person responsible for the product at his/her location, for either central or local integration. Companies with tool support use this by tagging at delivery. If there is also support for replication and such like, the receiver receives automatic access to the delivered documents. However, it is more common to, by ftp for example, manually copy to the receiver. Version control on the ftp server is obtained by following the pre-decided name conventions and file structures.

Ericsson's companies may use GASK, which is a product repository (central original repository) for frozen documents and products, other Ericsson companies who shall then use the released product can then simply fetch it from there. Together with PRIM, which is the central repository of the concern, including among other things all product revisions, one can by subscribing to the changes of a certain product revision, become aware of when a product is released. A good CM tool (including “MultiSite”) can give the same functionality.

Practical guidance

- All sites delivering a source code also must attach detailed installation instructions. If only binaries are delivered, the suppliers themselves must make sure that they can generate exactly the same binaries again. This also applies to deliveries between sites if these are subsequently used in the product.
- Define and describe the delivery process early in the CM plan. Deliveries and integrations are a large part of a distributed project and must therefore be well planned.
- Describe base line types applicable at important integration points. Let the CM plan indicate when and/or under which circumstances a base line is being made.

6.7 Consistent development environments

When developers are distributed, the development environment they work in are also distributed. Of course there will also be new versions of the environment which must be managed by the common CM management. The requirement is to know which version of each part of the development environment exists at each site. However, most people counter this by saying that there should be the same version at every site.

Different versions of a compiler may result in that what can be built at one site, cannot be built at another. However, the important thing is not that everything can be built at all sites but that it can be built in the target environment where the product is to be finally built, and that it can be used by the customers.

It is also important that one keeps and files the records on the correct environment required for building should it be required at a later point. One problem is that just knowing which versions of the different tools that have been used, may not actually be enough. In some operating systems, it is not always that easy to recreate the exact same environment by just installing the right versions of all applications. To be completely sure, a detailed description of the entire installation starting with an empty computer and the installation of the operating system is needed.

One issue that has to be considered is whether there should be a consistent development environment at the company level or at the project level.

Experiences

Ericsson Microwave has a user-type environment through which all systems and applications are started. Applications may not be started in any another way. In this way, one knows that everyone is running the same version, this also facilitates the installation of new versions. This environment exists for both Unix and PC (where it is called ESOE). When a new version of an application is bought, it is brought into the environment and the old version is eventually phased out. To direct the developers to use new versions, there are four different states for a version: (1) recommended (all new projects should have), (2) limited use (ongoing projects can continue to use), (3) being wound-up and (4) cancelled.

Another procedure is to allow different development environments, whilst still having only one environment in which everything is being built. The building environment is then put on a common server with a well-specified configuration.

If one has got a CM tool which supports replication, one can install all development tools at one site and let them be distributed automatically. If everything is gathered in this way, it is easy to store a snapshot of the environment at, for example, release and delivery.

Not using the latest version of a library in a delivered product may result in difficulties at maintenance. Old versions of the library are not always maintained by the supplier, and the newer (maintained) versions may not work with the rest of the product. It becomes particularly problematic if one has made patches oneself in the old version of the library and is now unable to upgrade.

Practical guidance

- Version control the whole development environment, by at least at release, freezing and storing the entire environment.
- Let all (important) applications be started via a user-type environment. Maintain that environment such that it is consistent between different development locations.

References

- [App98] Links to Software Configuration Management on the World Wide Web. <http://www.enteract.com/~bradapp/links/scm-links.html> Brad Appelton. 1999.
- [Bab86] Wayne A. Babich. Software configuration management : coordination for team productivity. Addison-Wesley. 1986. ISBN 0-201-10161-0.
- [Dar90] S. Dart. *Spectrum of Functionality in Configuration Management systems*. Technical report CMU/SEI-90-TR-11, Software Engineering Institute, Carnegie Mellon Institute, december 1990.
- [Fei91] P. Feiler. *Configuration Management Models in Commercial Environments*. Technical report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie Mellon Institute, mars 1991.
- [Fel79] S. I. Feldman, Make - A Program for Maintaining Computer Programs, *Software - Practice and Experience*, Vol 9, 255-265, 1979.
- [ISO95] Kvalitetsledning - Riktlinjer för konfigurationsledning. Standardiseringen i Sverige (SIS) SS-EN ISO 10 007.
- [Kel96] Marion Kelly. Configuration Management - The Changing Image. ISBN 0-07-707977-9. McGraw-Hill, 1996
- [Nym96] Ulf Nyman m fl, *Konfigurationshantering*. Sveriges Verkstadsindustrier, V040047, 1996
- [Roe75] Roekind, M. J., The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364-370, December 1975.
- [SEI95] *The Capability Maturity Model*. Software Engineering Institute, Carnegie Mellon University, Addison Wesley 1995.
- [TF94] Tichy, Walter F. och Feldman, S.I. (Eds.). Configuration Management. ISBN 0-471-94245-6. John Wiley & Sons. 1994.
- [Tic85] Tichy, W. F., RCS - a system for revision control. *Software-Practice and Experience*, 15(7):634-637, July 1985.
- [Tic88] Tichy, W. F., Tools for Software Configuration Management, *Proceedings of the International Workshop on Software version and Configuration Control*, Grassau, Germany, 1988.
- [Whi91] David Whigift. Method and Tools for Software Configuratioin Management. ISBN 0-471-92940-9. John Wiley & Sons. 1991.

Appendix - Tools and Interviews (in Swedish)

7 Forskning & Trender

En övergripande trend är att allt fler företag arbetar distribuerat. Utvecklingen går allt snabbare, pådriven av "time-to-market", vilket leder till större krav på parallell utveckling. Många har därför infört, eller ska snart införa, någon form av *verktygsstöd för distribuerad utveckling*. Då fler arbetar tillsammans trots den geografiska spridningen ökar behovet att olika platser ensar metoder, processer, verktyg och utbildning.

Från att företag tidigare mer eller mindre omedvetet hänna i olika situationer med geografiskt utspridda utvecklare är nu medvetenheten större. Nu planeras distribuerad utveckling i syfte att ha möjlighet till kraftsamling och för att bättre kunna utnyttja sina anställda, oavsett var de arbetar någonstans.

7.1 Forskning & Trender

I detta avsnitt tar vi upp några trender och forskningsområden relaterade till distribuerad utveckling och CM. Vi har delat upp dem i tre områden - fall av distribution.

Lokal utveckling och grundläggande funktionalitet

Mycket av CM-verktygens grundläggande funktionalitet och lagringstechnik har genomgått konsolidering. Tex används versionshantering av filer vilka lagras med delta-teknik av de flesta verktygen. Likaså används märkning (labels eller tags) för att sätta versioner på konfigurationer.

Några trender kan dock urskiljas:

- Komplexiteten för CM ökar p g a ett större antal objekt att konfigurationshantera. Detta beror i sin tur bl a på:
 - komponentuppdelning (vilket är en trend i sig). Allt fler företag delar upp sina produkter i komponenter, vilket ökar antalet objekt under CM.
 - användning av informationsobjekt istället för filer. Tex genom att konfigurationshantera enskilda element i SGML-filer.
- Livslängden på programvara blir allt kortare. Detta belyses bl a i VI-rapporten "Klarar era produkter Internet? ...när kunden vill styra, programmera och uppdatera dem via Internet" [Deg98]. Denna trend är en utmaning för CM.
- Inkrementell utveckling blir allt vanligare. Detta leder till att antalet konfigurationer och baselines kraftigt ökar, samtidigt som också

större krav på ändringshantering ställs.

- Change set-modellen har mognat och verktyg som stöder modellen finns på marknaden, t ex TRUEChange (se avsnitt 8.9). För att möta trenden stöder numera många icke change-set orienterade verktyg change-packages. Skillnaden mellan change-sets och change-packages diskuteras i avsnitt 8.9, Susan Darts artikel [Dar97] och [Wib97].

En forskningsgrupp i Trondheim ledd av Reidar Conradi [Conr98] utvecklade tidigt en prototyp där den grundläggande modellen är change-set.

Hur man kan versionshantera helheter, dvs sätta versioner på konfigurationer på samma sätt som på filer och hur strukturerade (hierarkiska) dokument kan hanteras forskar man på i Lund, Aalborg och i Aarhus, [Lund], [Aalborg], [Aarhus].

Fullständiga versioner sparar sällan i sin helhet utan endast i deltan, dvs skillnaden mellan nuvarande och förra versionen. Walter Tichy forskar bl a på algoritmer för att beräkna sådana delta för alla typer av filer (även binärer). Algoritmerna ska även så bra som möjligt upptäcka om delar av ett dokument flyttas, dvs det ska inte ses som en borttagning och en addering. [HVT96]

Conradi och Westfechtel har skrivit en översiktsartikel där olika versionsmodeller beskrivs [CW97].

Det bedrivs även forskning på hur processen ska stödjas av CM-verktygen. Exempel är verktyget EPOS (A Process-Centered Software Engineering Environment) [EPOS].

Sammanhållna grupper

De flesta verktyg har (eller får snart) funktionalitet för att replikera hela eller delar av arkivet på flera siter och att automatiskt synkronisera dessa. Enligt Ovum [BW98] tappar de leverantörer som endast har en central databas snabbt marknadsandelar.

Replikeringen i de kommersiella verktygen är sådan att alla grenar har en site som ägare. Strategin är konservativ så till vida att endast ägarsiten får skapa nya versioner på grenen.

I Lund [Lund] arbetar man med en mer optimistisk modell där dessa tekniska restriktioner inte finns.

Behovet av distribuerad utveckling och att snabbt kunna skapa distribuerade projekt leder till nya Internet-tjänster. Projektbibliotek sätts upp på "neutrala siter" vilka virtuella företag kan utnyttja för distribuerad utveckling. Den "neutrala siten" står för tekniskt stöd och tjänster för kommunikation, samarbetsprogram och delning av filer. Ett exempel på en sådan "neutral site" finns på SISU [SISU97].

Forskning för hur Internet-tjänster kan se ut och tekniskt implementeras bedrivs bl a på GMD i Tyskland [GMD].

Distribuerade grupper

Trenden för distribuerade grupper (avsnitt 2.1) är att situationen undviks men blir allt vanligare i takt med att också övriga fall av distribution ökar. I situationen med distribuerade grupper krävs speciellt stöd för

awareness som till viss del kan tillfredsställas med separata system, t ex LotusNotes, videokonferenser, samarbetsprogram (groupware), m m.

Forskning inom CSCW (Computer Supported Cooperative Work) ger allt bättre stöd för synkron/samtidig interaktion. Olika modeller tas fram för interaktion mellan användarna. Tex synkron kommunikation då alla samtidigt ändrar i ett dokument, eventuellt med WYSIWIS (What You See Is What I See), eller asynkron kommunikation då användarna vid olika tidpunkter ändrar i ett gemensamt dokument. [CSCWyp]

Separata verktyg för interaktion ger en tyngre miljö att arbeta i och kräver kopiering av data mellan verktyg. En integration av sådan funktionalitet med ett CM-verktyg skulle göra det lättare att använda.

I Lund [Lund] arbetar man med en integrerad utvecklingsmiljö med stöd för awareness. Målet är att i samma modell både ge stöd för synkron- och asynkron kommunikation för att på så sätt kunna stödja de krav på awareness olika faser i ett projekt har. [MM93]

7.2 Konferenser och journaler

För den som vill följa forskningen inom området CM finns idag ett par större konferenser, deras proceedings samt vetenskapliga tidskrifter att tillgå. Av konferenserna är "System Configuration Management" (SCM) den mest centrala. Området fick ett förnyat intresse när SCM startades 1988 och konferensen har anordnats 8 gånger, på senare tid varje år. Proceedings publiveras i Springer Verlags serie "Lecture Notes in Computer Science" och är relativt lätt att få tag i, t ex [SCM-6,7,8,9]. Vetenskapliga resultat i CM kan också hittas från de generella Software Engineering konferenserna; "International Conference on Software Engineering" (ICSE), t ex [ICSE97], "ACM SIGSOFT International Symposium on the Foundations of Software Engineering" (FSE), t ex [FSE6] och "European Software Engineering Conference" (ESEC).

På samma sätt förekommer resultat från CM-området i tidsskrifter som "IEEE Transactions on Software" och "IEEE Transactions on Software Engineering".

Det finns även en grupp forskare inom CSCW-området (Computer Supported Cooperative Work) som intresserar sig för programvaruutveckling. Exempel på knypunkter mellan disciplinerna är hur awareness (vilket är ett begrepp från CSCW-området) kan presenteras för användare i en utvecklingsmiljö. Ett exempel på konferens är "European Conference on Computer Supported Cooperative Work" (ECSCW), t ex [ECSCW97]. "Computer Supported Cooperative Work - An International Journal" [CSCW], är en tidsskrift i området.

www-länkar

Två sidor med många ytterligare länkar är:

- Configuration Management Yellow Pages: http://www.cs.colorado.edu/users/andre/configuration_management.html [Hoe99]
- Brad Appletons länkar inom CM-området: <http://www.enteract.com/~bradapp/links/scm-links.html> [App98]

8 Verktyg

Vi skall här försöka redogöra för hur en del av de verktyg som finns på marknaden behandlar och löser problem relaterade till distribuerad utveckling. Ingen fullständig utvärdering görs, utan vi fokuserar på de områden vi behandlat tidigare i denna rapport.

För att kunna ge vägledning åt så många olika typer av företag som möjligt är urvalet av verktyg gjort sådant att både "low-end" och "high-end" verktyg är med, där uttrycken "low-end" och "high-end" definieras i termer av den funktionalitet som ingår. Dessutom har en spridning i prisklass eftersträvats. Ett mål har också varit att välja verktyg så att de synkroniseringsmodeller som beskrivs i rapporten är representerade.

För varje verktyg ges först en kort sammanfattande beskrivning av funktionaliteten. Där ingår bl a vilken grundstrategi som verktyget följer och vilken synkroniseringsmodell som i huvudsak stöds.

Informationen är mestadels hämtad från författarens och referensgruppens erfarenheter, samt från intervjuerna, men "rådata" är även hämtad från bl a verktygsmanualer, Ovums utvärderingar av CM-verktyg [RBI95, BGD96, BW98], Susan Darts artikel [Dar90] och Peter Feiler artikel [Fei91].

Obs! Beskrivningarna är *inte* en fullständig jämförelse mellan verktygen. Ingen gemensam testsituation har använts och beskrivningarna är gjorda av olika personer med olika erfarenhet av verktyget. Syftet är att ge en kort beskrivning av verktygets själ och en fingervisning på vilket sätt verktyget stöder distribuerad utveckling.

Ett samlat intryck är att man inte får se ett verktyg, oavsett vilket man väljer, som lösningen på alla problem. Generellt sett är det i stället så att man måste veta lösningen *innan* verktyget kan hjälpa till och att en egen väl utvecklad process och enklare verktyg kan vara det mest kostnadseffektiva och det som passar den egna organisationen bäst.

De verktyg som tas upp är:

- ClearCase – ett av de stora heltäckande verktygen. Versionsbaserat.
- Continuus – ett av de stora heltäckande verktygen. Processorienterat.
- CVS – freeware med bra funktionalitet.
- ExcoConf – svenskt folderbaserat verktyg utvecklat med stöd från Ericsson.
- JavaSafe – ett verktyg kopplat till det relativt nya programmeringspråket Java.
- PCMS – ett av de stora heltäckande verktygen. Bäst i Ovums senaste utvärdering.
- PVCS – bland de största på PC-marknaden.
- Teamware – stöder helt synkroniseringsmodellen Långa transaktioner.
- TRUEchange – stöder helt synkroniseringsmodellen Change set.

- Visual SourceSafe – intressant p g a sin marknadsspridning.

8.1 ClearCase

ClearCase [Clear98, BW98, AM97, Cla95] är ett versionsorienterat, klient/server-baserat verktyg som stödjer alla CM-aspekter - versionshantering, ändringshantering, konfigurationshantering, osv.

ClearCase produktfamilj består av:

- ClearCase - CM-verktyg med versions- och konfigurationshantering samt stöd för effektiv generering.
- ClearCase MultiSite - verktyg som replikerar information på flera siter.
- ClearQuest - Felhanteringssystem.
- ClearCase Attache - Add-on som erbjuder en delmängd av ClearCase funktionalitet för Windows-plattformarna.
- ClearGuide - processtöd genom att kunna definiera arbetsuppgifter, prioritera aktiviteter, allokerar resurser och följa projekt.

ClearCase är mest intressant för:

- medelstora och stora utvecklingsprojekt i Windows eller Unix.
- företag med distribuerad utveckling.

Versionshantering

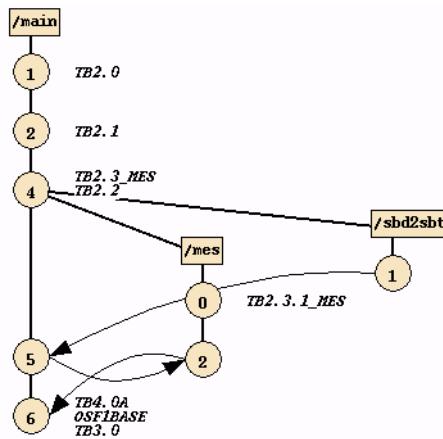
ClearCase versionshanterar alla typer av filer och kataloger. Nya versioner skapas genom ut- och incheckning. Grenar kan skapas och senare ensas (merge). Versioner av konfigurationer hanteras genom att sätta en label på alla ingående filer och kataloger.

Figur 14 visar ett grafiskt användargränssnitt för att presentera en fils versionshistoria. Två nya grenar (sbd2sbt och mes) har skapats från huvudgrenen (main). Grenarna har senare ensats (genom merge) med huvudgrenen. Några av versionerna är märkta med en eller flera labels, t ex så är version 1 i huvudgrenen märkt med labeln TB2.0. Genom att markera en version i det grafiska gränssnittet kan operationer utföras på filen (t ex checka ut). Genom att välja flera versioner kan skillnader presenteras och grenar ensas (diff- och merge-verktyg startas med de valda versionerna som parametrar).

All versionshanterad data lagras i arkiv, vilka i ClearCase-terminologi heter VOB:ar (Version Object Base). Dessa innehåller katalogstrukturen och en databas i vilken referenser, länkar och metadata lagras. För att spara diskutrymme lagras deltan (skillnaden mellan två versioner) i VOB:en i stället för det fullständiga innehållet av alla versioner.

Konfigurationer/Urval

Allt arbete i ClearCase sker i s k vyer (view). För att kunna komma åt (accessa) filer och kataloger måste en vy vara satt. Vyn fungerar som ett filter vilket bestämmer vilken version av en fil användaren ser när han/hon väljer filen genom dess namn. När väl en vy är satt, kan hela kata-

**Figur 14** Versionsgrafen för en fil.

logträdet i VOB:en ses som om det vore i det normala operativsystemet (dvs med "ls" och "pwd" i Unix, och med t ex utforskaren i Windows).

Till en vy hör en s k konfigurationsspecifikation, se figur 15. Specificationen består av regler vilka evalueras en efter en tills en version av filen hittas som passar in på regeln.

Vyer erbjuder på så sätt ett transparent användargränssnitt, vilket gör att alla de vanliga verktygen som hanterar kataloger och filer kan användas. Användarna (och verktygen) kan referera till filerna på det vanliga sättet genom att endast ange dess namn. Reglerna i vyn bestämmer vilken version av filen som hämtas.

En vy har dessutom en arbetsyta. När en utvecklare checkar ut en fil läggs en arbetskopia av filen i den aktuella vyn. När filen senare checkas in skapas en ny version i VOB:en vilket gör den åtkomliga för de övriga utvecklarna. Alla genererade filer läggs också i vyns arbetsyta, men kan även dessa checkas in i en VOB, t ex för att arkivera en frisläppt version.

Parallell utveckling

Parallelt arbete stöds genom möjligheten att skapa grenar. Olika utvecklingsstrategier kan åstadkommas genom olika regler i konfigurationsspecifikationen. Eftersom reglerna evalueras vid varje fil-access kan ändringar i VOB:en få genomslag för andra utvecklare redan vid deras nästa access av den ändrade filen (optimistisk uppdateringsstrategi). Genom att arbeta i olika grenar och använda regler som väljer ut versioner från de olika grenarna kan mer komplicerade strategier erhållas. Det finns dock en risk för att användarna inte längre förstår hur urvalet

```

element * CHECKEDOUT
element * /main/mes/LATEST
element * /main/TB2.3_MES

```

Figur 15 Regler i en konfigurationsspecifikation. De två översta (första) reglerna är generiska och resulterar i olika versioner allteftersom in- och utcheckningar görs. Den nedersta regeln resulterar alltid i samma version, märkt med etiketten

görs och därfor inte upptäcker i fall någon regel skulle vara fel och fel versioner används. Exempel på olika strategier är:

- En konservativ uppdateringsstrategi erhålls genom att, förutom sina egna utcheckade, endast välja ut specifika versioner (t ex en viss version eller de versioner märkta med en viss label).
- En optimistisk uppdateringsstrategi erhålls genom att använda regeln LATEST, vilket innebär att den senaste versionen på den angivna grenen väljs. Om någon annan utvecklare checkar in en fil kommer den nya versionen väljas nästa gång reglerna evalueras, dvs vid nästa access av filen.
- Riktigt optimistisk blir strategin om flera utvecklare arbetar i samma vy. Då kommer de även att använda samma utcheckade filer.

För ensning av grenar används vad de kallar 32-vägs merge (som 3-vägs merge men fler än två grenar kan mergas åt gången). ClearCase hittar själv den senast gemensamma versionen och beräknar de ändringar som gjorts från den gemensamma versionen och de versioner som ska ensas. Ensningen är radbaserad, dvs hela rader hanteras snarare än faktiska ändringar.

Distribuerad utveckling

ClearCase MultiSite gör det möjligt att replikera hela VOB:ar på flera siter. Replikeringen kan ske automatiskt vilket innebär att alla nya versioner som checkats in på en site även blir synliga på de övriga. Detta gör, enligt Ovum [BW98], ClearCase marknadsledande inom området "Remote development".

Varje gren i VOB:en har en ägare, en site. Endast ägar-siten får skapa nya versioner på grenen. De övriga kan endast läsa, eller skapa nya undergrenar. På så sätt riskerar man aldrig att någon ensning av innehållet i en fil (merge) måste göras vid synkroniseringen av två siter.

Lösningen innebär ett bra stöd för fallet Sammanhållna grupper (avsnitt 2.1). För utvecklare i samma grupp och som normalt arbetar på samma gren kan det dock vara svårt att arbeta distribuerat, men en möjlig lösning är att varje person får en egen gren.

Övrigt

I ClearCase ingår också bygg-verktygen Clearmake (Unix) och Omake (PC). Dessa återanvänder redan genererade filer då detta är möjligt, även mellan olika vyer och siter. De skapar också en lista ("bill of material") över alla de i det byggda systemet ingående filer och deras versioner, tillsammans med versionerna för de under genereringen använda verktygen.

8.2 Continuus

Översikt

Continuus [Con98, BW98, Cla95] är ett processdrivet, klient/server baserat verktyg som stödjer alla CM-aspekter - versionshantering, ändringshantering, konfigurationshantering, osv. Serverdelen är implementerad på plattformarna Unix och Windows NT, och klientdelen på Unix, Windows NT och Windows 95. En Informix databas används för datalagring.

Continuus består av flera komponenter:

- Continuus/CM - CM-verktyget som innehåller stöd för CM-processmodeller. Det har stöd för parallell och distribuerad utveckling.
- Continuus/PT - Felhanteringssystem fullt integrerad med CM-verktyget.
- ObjectMake - Make-kompatibelt verktyg som stödjer distribuerat byggande
- WebSynergy Content - ett Web-baserat verktyg för hantering av stora Internet och intranet siter.

Continuus är mest intressant för:

- Medelstora och stora företag som vill ha en full kontroll över CM-processerna.
- Företag med intensiv distribuerad utveckling.
- Utveckling och underhåll av stora Web och intranet siter.

Versionshantering

Continuus hanterar objekt, vilka kan bestå av källkods- eller exekverbara filer, kataloger, dokument, datafiler, osv. Alla objekt är under versionskontroll och nya versioner av objekten skapas när de checkas ut och in. Parallella versioner av objekten kan skapas.

En grafisk versionshanterare visar utvalda objekt med egenskaper för varje version. Dessutom går det att jämföra två versioner och slå samman versioner (merge).

Continuus lagrar objekt (projekt, filer och kataloger) i en arbetsstruktur (work space) på användarnas lokala diskar. Användarens arbetsstruktur är en projektion av en eller flera databasstrukturer. I Windowsmiljön innehåller strukturen kopior av filer som finns i databasen, medan Continuus i Unixmiljö i stället använder sig av symboliska länkar. Man checkar ut ett objekt om man vill modifiera det. I en distribuerad miljö uppdateras lokala arbetsstrukturer automatiskt eller på begäran av användaren.

Man kan använda Continuus även när man är främkopplad från systemet (till exempel med en bärbar dator). Vid uppkopplingen görs en synkronisering med systemet.

Parallell utveckling

Continuus skiljer på tre olika typer av parallell utveckling som beskrivs nedan: (1) Samtidig (concurrent) utveckling, (2) utveckling av parallella

produktversioner (temporära grenar) och (3) utveckling av olika varianter av produkter (med permanenta grenar).

1) Samtidig utveckling sker när flera utvecklare samtidigt inför förändringar i olika versioner av samma objekt. Dessa förändringar måste slås samman innan man bygger upp en ny produktversion.

2) Parallel versionshantering av produkter har man när man utvecklar och underhåller flera temporära grenar av en produkt samtidigt, t ex bugg-fix och utvecklingsvarianter. Vissa ändringar genomförda i en gren vill man även införa i någon annan gren.

3) Varianthantering inkluderar stöd för utveckling av olika permanenta grenar av samma produkt (till exempel samma produkt för olika plattformar). Man använder olika attribut, som bygg-flaggor, och olika varianter av objekten, men man slår aldrig samman dem.

För att slå samman olika versioner av ett objekt använder man ett Merge Wizard program.

Distribuerad utveckling

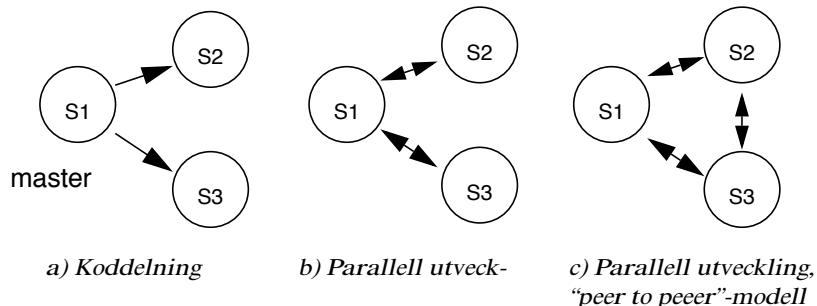
Continuus har ett mycket bra stöd för distribuerad utveckling. Distributed Code Management (DCM) är en tilläggskomponent som hanterar distribuerade databaser. Man skiljer på två olika typer av distribuerad utveckling (Figur 16):

- Kodelning - En site (master site) är ansvarig för all ändring av koden vilken används på andra siter (a).
- Parallel utveckling - alla siter kan jobba med samma objekt och DCM uppdaterar ändringarna på alla siter (b och c).

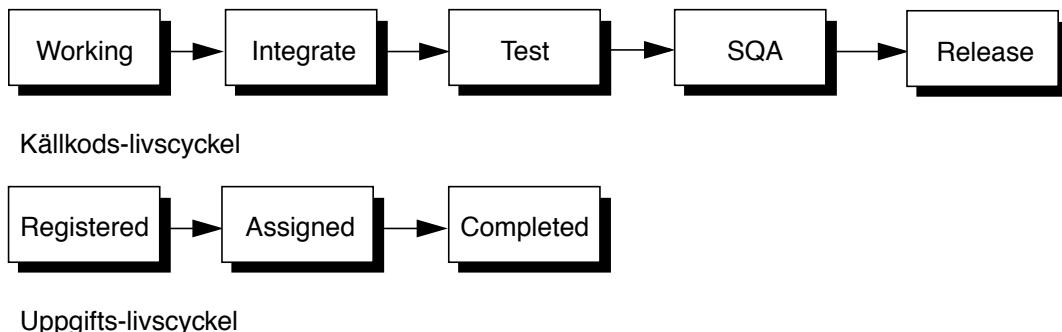
DCM kan skicka en komplett databas, databasdeler eller enstaka objekt mellan olika siter.

Ändringshantering och processhantering

Det finns stöd för ändringshantering och för felhantering som är bra integrerade med olika CM-funktioner. Continuus/PT, tillägg till grundpaketet, är ett felhanteringssystem. Det stödjer kopplingen mellan felrapporter och ändringsbegäran eller *uppgifterna* (tasks). Uppgifterna är i sin tur kopplade till versioner av objekt. Dessutom innehåller



Figur 16 Continuus distributionsmetoder



Figur 17 Standardlivscykeln för källkodsfiler och uppgifter

uppgifterna information om ansvariga utvecklare, planerat och utfört arbete, osv. Med hjälp av uppgifterna kan man få en bra översikt över projektet.

Continuus har stöd för processhantering. Man kan definiera en livscykel för utvecklingsprojekt, källkodsfiler och uppgifter. Varje tillstånd i cykeln definieras av ett antal attribut och möjliga övergångar till nästa tillstånd. För att definiera en modell för en livscykel, använder man Continuus ACCent kommandospråk.

Figur 17 visar standardlivscykeln för olika objekt/processer i Continuus.

Integrering med andra verktyg

Continuus använder Microsoft SCC Interface som gör det möjligt att ha en integrering med Microsoft Developer Studio (Visual C++, J++, Visual Basic) på exakt samma sätt som t ex Microsoft Visual SourceSafe har.

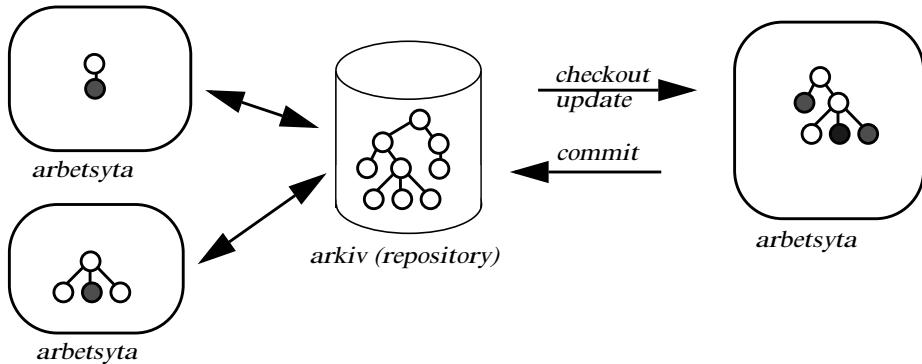
Continuus innehåller en Web-klient som kan användas via Internet. Alla Continuus/CM och Continuus/PT funktioner utom vissa administrativa funktioner finns inbyggda i Web-klienten.

8.3 CVS

Allmän översikt

CVS är en gratisprogramvara under GNUs avtal om spridning och försäljning. Det verkar dock vara många som arbetar med koden och som säljer tjänster i form av support och utbildning av CVS. Verktyget är portat till de flesta Unix-plattformarna, NT, OS/2 och VMS. Exempel på webbsidor med mer information är [Cyc99, Mol98, Ced93].

CVS (Concurrent Versions System) versionshanterar hela katalogträdet i filstrukturen (snarare än enskilda filer som t ex RCS gör). Standardkonfigurationen är ett centralt källkodsarkiv (repository) mot vilket alla arbetar. Varje utvecklare har en egen arbetsytan (working directory) där det egna arbetet utförs och till vilken enstaka filer, hela katalogträdet eller s k moduler checkas ut. Moduler definieras av användarna själva och består av en mängd filer eller kataloger. CVS definition av kommandot checkout innebär endast kopiering av de berörda filerna till arbetsytan.



Figur 18 Centralt arkiv och lokala arbetsytter till vilka hela katalogträd är utcheckade

Ingen låsning sker utan andra utvecklare kan fortfarande checka ut samma filer. I den lokala arbetsytan kan sedan alla typer av modifieringar göras (editera, skapa eller ta bort filer och kataloger), utan att det centrala arkivet påverkas. När den logiska ändringen är slutförd integrerar man tillbaka de ändrade filerna/katalogerna till arkivet. Dvs den arbetsprocess som stöds är kopiera-modifiera-merge. Det går också att när som helst uppdatera sin arbetsyta med de förändringar som gjorts i arkivet sedan man sist checkade ut/uppdaterade sin arbetsyta.

Figur 18 visar ett arkiv och tre arbetsytter till vilka olika delmängder av det totala arkivet har checkats ut. Eftersom (normalt) ingen låsning sker vid utcheckningen kan samma filer (i samma version) checkas ut till flera arbetsytter. I arkivet skapas nya versioner av filerna vid incheckning (commit), vilket gör att man alltid kan backa till en tidigare incheckad version. I arbetsytorna sker, emellertid, ingen lokal versionshantering.

Om en fil har modifierats i arkivet (ny version har skapats) sedan filen checkades ut till arbetsytan kommer en commit-operation att avbrytas. Integrering måste i stället först göras i arbetsytan genom update och merge av berörda filer. Själva ihopslagningen görs automatiskt ("icke-interaktivt") av systemet och resultatet lagras i arbetsytans version av filen. Standardbeteendet är att slå samman ändringarna så länge de inte är i konflikt (dvs överlappar). Om de överlappar så får man båda versionerna i sin arbetsfil, speciellt markerade (vilket alltså för källtexter ger en icke-kompilerbar fil).

Praktiskt sett så fungerar merge-reglerna bäst för textfiler där ordningen i filen inte kastas om mellan editeringar, exempelvis källtexter, make-filer och textfiler med löpande text. Dvs inte för binärfiler och automatgenererade (text)filer där ordningen kan kastas om radikalt. Det är också möjligt att i CVS markera en fil som "binär". Vid merge görs då inget försök att ensa filernas innehåll, utan i stället väljs den senaste versionen av de två.

Exempel på övrig funktionalitet är grenar (branch) och "taggar" vilket ger stöd för parallell utveckling respektive att skapa versioner av konfigurationer. Triggers erbjuder (svagt) stöd för processhantering och ändringshantering. Arbetsytorna gör att man isolerat och utan att hindra andra utvecklare kan göra många relaterade ändringar och sedan testa dem

lokalt innan man uppdaterar det gemensamma arkivet. Efter commit har man dock ingen spårbarhet kvar om vilka filer som ändrades ihop. För att erhålla detta krävs att man märker dem med samma "tag" efter commit.

Distribuerad utveckling

Exempel på arkitekturen med CVS är:

- Ett centralt arkiv mot vilken alla arbetar via ett nätverks filsystem (networked filesystem). Fungerar bra om nätverket är bra. Detta är standardkonfigurationen av CVS.
- Client/server CVS, dvs ett centralt arkiv mot vilket klienter arbetar via CVS egna fjärrprotokoll (CVS remote protocol). Brukar i CVS dokumentation även kallas att de stöder "network transparency". Kan användas om nätverket är osäkert, t ex över Internet eller modem. CVS egna protokoll är effektivt och robust. Om förbindelsen bryts mitt i ett kommando läses inte servern för de övriga utvecklarna.
- Multi-site med flera arkiv; en master och resten slavar. Man gör all incheckning mot mastern men man kan även checka ut från slavarna vilka automatiskt hålls uppdaterade (det finns olika generella speglingsverktyg som kan hantera detta t ex rdist, rsync eller CVSUp). Arkitekturen motsvarar närmast den i avsnitt 5.4, "Several sites by Master-Slave connections", men med högre frekvens på uppdateringen.

Vid sekretesskrav finns det möjlighet att använda Kerberos-protokoll för identifiering av fjärranvändare.

I senare versioner av CVS finns nu även stöd för awareness via s k watchers. Om man aktiverat en watcher på en fil blir den endast läsbar vid utcheckning (och inte skrivbar som annars är normalt). För att få skrivrättigheter behöver man göra ytterligare ett kommando, "edit", vilket då också skickar ut ett meddelande om detta till alla som prenumererar på filförändringar för just den filen. På så sätt kan man, för en delmängd intressanta filer, hålla sig underättad om vad som händer. Ett exempel på intressant filtyp är "icke-mergebara" filer, som med en kombination watch/edit och binärfilsmarkering kan få en "rimlig" hantering. För filer eller hela katalogträd kan man:

- begära "notification" när någon börjar/slutar editera (edit/commit, unedit)
- få en lista över de utvecklare som just nu arbetar med en fil
- få en lista över de utvecklare som aktiverat watch på en fil.

Jämförelser

Jämfört med den vanliga checkout/checkin-modellen ger CVS ett bättre stöd för att hantera hela katalogträd med filer. Modellen blir därmed mer lik långa transaktioner genom att man under arbetet ändrar de filer man vill utan att behöva CM-hantera dem en och en. Vid commit tar sedan systemet reda på vilka filer som verkligen behöver mergas tillbaka.

Stöder dock inte transaktionsmodellen helt då man inte har någon intern versionshantering i arbetsytan (vilket t ex Teamware har).

CVS lösning med flera siter erbjuder inte samma automatiska synkronisering som t ex ClearCase Multi-Site gör. Lösningen med en master och flera slavar blir snarare ett ytterligare stöd för långa transaktioner. En slav kan ses som en subtransaktion till mastern (fadertransaktionen) i vilken man isolerat kan arbeta under lokal versionshantering. När man är färdig checkar man in till mastern vilket motsvarar commit till fader-transaktionen.

8.4 ExcoConf

Allmän beskrivning

ExcoConf [Exco99] är ett svenskt verktyg från början utvecklat av Ericsson, men utvecklas nu av företaget ExcoSoft.

ExcoConf hanterar folderstrukturer och är anpassat till Ericssons principer för produkt- och dokumenthantering.

Verktyget finns för ett antal datorsystem som Windows, Unix (HP-UX, Solaris) och VAX/VMS. ExcoConf ser inte ut som ett modernt Windows-program speciellt vad gäller dialog för val av filer, arbetskataloger m m. Verktyget har samma användargränssnitt oberoende av vilken plattform som används.

ExcoConf levereras inte med några tilläggsfunktioner för vare sig ärendehantering eller jämförelse mellan dokument av olika versioner. Verktyget koncenterar sig helt på grundfunktioner för hantering av komplexa folderstrukturer.

Varje version av en fil lagras som en "hel" fil, dvs ingen deltalagring. Operativsystemets behörighetskontroll bestämmer vilka som får tillgång till olika foldrar och dokument.

Funktionen för checkout/checkin har lösts på ett annorlunda sätt jämfört med de andra verktygen. Om man som konstruktör skall modifiera en komponent i en folder gör man en ny version av foldern som från början är tom. För de komponenter (filer) som skall modifieras ger man ett speciellt kommando för uppdatering så att filen hamnar i ett arbetsläge, dvs i den nya ännu ej frusna versionen av foldern. Därefter kan den uppdateras tills man fryser foldern. Detta ger en arbetsmodell lik den som beskrivs i avsnitt avsnitt 4.3, "Long transactions".

Andra utvecklare kan se när en ny version av en folder skapas och de kan också se de ändringar som görs i foldern.

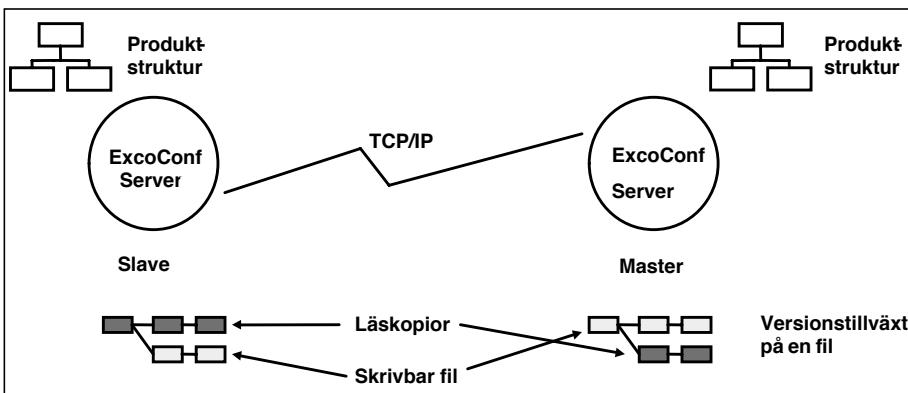
ExcoConf stödjer parallell utveckling genom att man definierar grenar (brancher) på en folder, som sedan kan ensas (merge).

Distribuerad utveckling

Tre av de arkitekturen vi presenterat tidigare i rapporten stöds idag av ExcoConf Multisite. Hur dessa stödjs, beskrivs övergripande nedan.

Lokalt mot en server

Då arbetet sker lokalt mot en server äger ExcoConf-servern databasen, dvs alla filer. Man kan sätta upp behörighet på alla användare, som skall



Figur 19 Två servrar, en master och en slav. Mastern äger huvudgrenen medan slav-servrern i det här fallet äger den andra grenen.

få göra update/release på foldrar. Operativsystemets behörighetskontroll bestämmer vilka som får tillgång till olika dokument.

Bärbar dator mot en server

Arbete med bärbar dator stödjs på så sätt att man kopierar ut det man vill arbeta med på sin bärbara PC. När man ska lägga tillbaka informationen gör man det genom ett "restore"-kommando. Arbetssättet fungerar bra om strukturen redan finns på PC'n. Om det inte finns en struktur, måste användaren först skapa den, innan han/hon kopierar ut den aktuella informationen.

Det är filer i arbetsläge som kopieras ut och kopieringen sker utan att några lås sätts på filerna, dvs det är möjligt att parallellt även modifiera filerna i arbetskatalogen. Andra utvecklare kan heller inte se att utkopiering har skett. Tillsammans ger detta en risk för att filer ofrivilligt och oavsiktligt ändras parallellt.

Flera siter med olika ansvarsområden

Fungerar enligt den arkitektur som beskrivits tidigare i avsnitt 5.5, "Several sites with differing areas of responsibility".

Arkitekturen används av EMW (se intervju i avsnitt 10.7) i ett fall för att föra över information mellan olika datorplattformar. I just det fallet är det mellan VAX/VMS och Unix.

Ett Ericssonbolag i Stockholm använder MultiSite-funktionaliteten mellan tre olika siter, Sverige, Finland och Tyskland.

ExcoConf har möjlighet att sättas upp som en Master-server med en eller flera slave-servrar. Man anger var servern finns, vilka andra servrar som får accessmöjligheter samt vilken av servrarna som skall anses vara Master-servern. Systemadministratören ser sedan till att Slave-serverna startas upp med den multisite-menyn som finns. För att underlättा användandet för användarna, kan systemadministratören se till att en projektarea kopplas till.

I figur 19 visas ett exempel på hur Multisite fungerar. Två ExcoConf-servrar finns uppsatta med en kommunikation enligt TCP/IP. Folder-

strukturen som Mastern har äganderätten till, kommer att visas på Slav-servern, men enbart med läsrättigheter.

Om en konstruktör vill uppdatera en folder på Slav-serversidan måste en gren skapas i vilken han/hon då kommer att ha skrivrättigheter. Motsvarande gren kommer att synas på Master-servern, men enbart med läsrättigheter, dvs en lokal kopia. Dubbel lagring av folder och dokument behövs. För att se om något har hänt på Master-servern, t ex något nytt dokument eller ny version har skapats, måste man manuellt begära uppdatering på den egna Slav-servern.

Master kan när som helst lämna över sitt masterskap till någon slav-server.

ExcoConf Multisite har en synkroniseringsfunktion, som t ex kan köras varje natt för utbyte av information mellan servrar.

8.5 JavaSafe

Java är ett relativt nytt programmeringsspråk som, åtminstone i början, mest marknadsförts som ett plattformsberoende språk och därfor väl lämpat för Internet, t ex för att få Java-applets på web-sidorna. JavaSafe från Sun, är inte ett fullständigt CM-verktyg utan mer ett verktyg för att hantera Java-källkod. Det kan dock vara intressant att se vad ett nytt verktyg (version 1.0) för ett "Internet-språk" kan erbjuda.

Enligt [JS98] erbjuder JavaSafe ett Javabaserat, plattformsberoende, versionshanteringssystem för att koordinera geografiskt distribuerade grupper vars arbete är att modifiera komplexa programvarukonfigurationer och binära system.

Verktyget har en klient/server-arkitektur med en server mot vilka alla klienter arbetar. Det är framförallt checkout/checkin-modellen som stöds, men två olika typer av lås ger valfrihet. Grenar kan skapas och ensas (merge). Alla typer av dokument kan hanteras, t o m web-innehåll.

Allt arbete sker i arbetsytor i klienterna. Filer checkas ut från arkivet till en arbetyta och modifieras där. Vid incheckning skapas en ny version i arkivet.

Synkroniseringsmodeller

Två olika typer av lås finns: exklusivt och delat. Om exklusivt lås används innebär det att endast en användare i taget kan checka ut en version av en fil, och synkroniseringen sker således enligt checkout/checkin-modellen. Delat lås innebär att flera utvecklare samtidigt kan checka ut från samma version till sin arbetsyta, alla med delat lås.

Tillsammans med stöd för att synkronisera arbetsytor och arkivet stöds även synkronisering enligt långa transaktions-modellen. Om en användare checkar in en fil får de övriga användarna som checkat ut filen med delat lås ett meddelande om att deras version inte längre är aktuell och att de bör uppdatera sin arbetsyta. Detta tillför en awareness i modellen som t ex Teamware saknar.

| lineup, version 1 | lineup, version 2 |
|---------------------------|---------------------------|
| <i>Fil A, version 1.1</i> | <i>Fil A, version 1.2</i> |
| <i>Fil B, version 1.1</i> | <i>Fil B, version 1.2</i> |
| <i>Fil C, version 1.1</i> | <i>Fil C, version 1.1</i> |

Figur 20 Versioner av lineups i arkivet

Konfigurationer

Konfigurationer stöds på två sätt. Dels kan användaren skapa egna "kartor" av arkivstrukturen, dvs vyer eller grupperingar av data lagrad i arkivet. Dessa "kartor" kan "frysas", vilket motsvarar en "taggning" av ingående filer.

Dessutom versionshanteras s k "lineups". Varje aktuell konfiguration i arkivet är en lineup. Figur 20 visar två versioner. Version 1 består av tre nyskapade filer: A, B och C. Fil A och B checkas ut och modifieras. När de checkas in igen skapar JavaSafe en ny version av arkivets lineup i vilken de nya versionerna ingår. Sambandet mellan ändringarna i fil A och B bibehålls således även efter checkin. Gamla versioner sparas och det går alltid att återskapa filer som de var i tidigare lineups.

Distribuerad utveckling

Arkitekturen är en server med ett globalt arkiv. Inget stöd ges för att replicera och synkronisera arkiv.

Övrigt

Stöd ges för att skapa listor (bills) för vad som ingår i en frisläppning; både dokument, runtime-miljö, utvecklingsmiljö, m m.

För att implementera processtöd erbjuds "hooks" (även kallade triggers), vilket är subrutiner som körs vid specificerade tidpunkter, t ex före och/eller efter checkin.

Mer information om JavaSafe kan hittas på Suns web-sidor, bl a [JS98].

8.6 PCMS

Anmärkning: PCMS [PCMS99] bytte namn till PVCS Process Manager, när produkten köptes av Intersolv (ej att förväxla med PVCS, vilket är ett annat verktyg med samma ägare, se avsnitt 8.7). Namnet PCMS används i texten eftersom produkten är mest känd under det namnet

Allmänt

PCMS är ett processorienterat CM verktyg som stödjer alla faser av konfigurationsledning. Funktionaliteten spänner över enkel versionshantering enligt CheckIn/CheckOut modellen till ett fullständigt konfigurationshanteringssystem med integrerade funktioner för ändringshantering, baselines, byggnation och release av programvara.

PCMS kan, beroende på hur det används, mer eller mindre stödja samtliga transaktionsmodeller som beskrivits tidigare i rapporten. Tynghpunktens ligger dock åt change set modellen, eftersom den integrerade ändringshanteringen spelar en central roll i verktyget.

PCMS stödjer även livscykler, dokumenttyper, roller, mallar och ändringsdokument vilka alla kan anpassas av användaren. Namn på statustyper och roller för alla livscykler (processer) är definierbara av användaren. Detsamma gäller för attributnamn (metadata) på objekt. Alla metadata lagras i en Oracle databas vilket ger bra möjligheter att skapa egna rapporter.

PCMS finns förutom på de vanliga datorplattformarna även på VMS. Alla plattformar kan tillsammans köras mot en gemensam databas i ett heterogent nätverk.

Sekretess

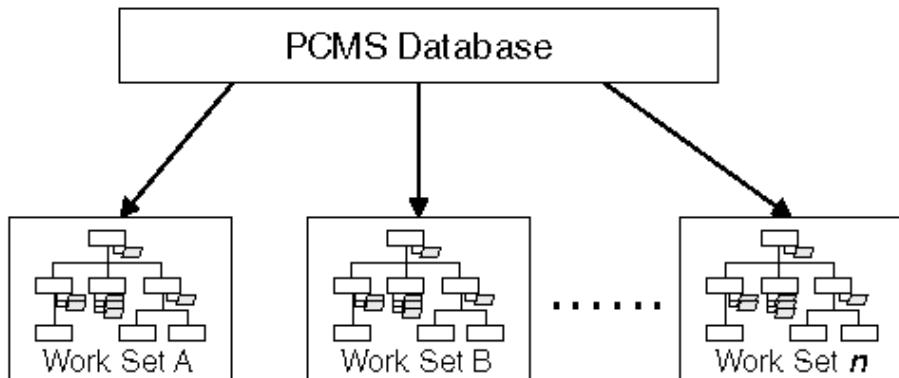
I PCMS går det lätt att styra åtkomsten till filer och metadata. Genom att tilldela roller på olika platser i produktstrukturen kan man fördela rättigheter. Vill man sedan ha högre upplösning så kan man också definiera roller per workset (vy). Eftersom man sedan kan styra vad ett workset ska innehålla går det alltså att i en produkt ha t ex ett workset som bara innehåller ett dokument, och bara har sig tilldelat en enda roll. Det är då bara den personen som kan läsa dokumentet. Rättigheter kan också knytas till vilken status i livscykeln ett objekt befinner sig.

Parallel utveckling

PCMS använder sig av workset för att definiera den omgivning som varje användare arbetar i. Ett workset är en vy mot CM databasen. Den vyn kan innehålla hela produkten eller valfria delar av den, t ex ett delsystem. Ett workset kan skapas tomt eller från en baseline. Roller kan tilldelas per workset.

Exempel (figur 21): Release 1.0 av en programvara har levererats till kund och arbetet fortsätter med att utveckla release 1.1. Detta arbete sker inom workset A.

Kunden rapporterar problem med version 1.0 av programvaran. För att rätta problemet utan att störa utvecklingen, skapas workset B utifrån



Figur 21 Utveckling och underhåll i work sets.

release 1.0, dvs innehållet i workset B motsvarar exakt innehållet i release 1.0.

De ändringar (nya revisioner av filer) som genomförs i workset B, är helt skilda från workset A. När ändringarna är klara kan en ny baseline fastställas och patch release 1.01 skickas till kund. Workset B kan nu tas bort. Om ytterligare ändringar behöver göras så kan ett nytt workset enkelt skapas från antingen release 1.0 (annan kund) eller från release 1.01 (rättning åt samma kund som innan).

Normalt vill säkert utvecklingsteamet veta vilka problem som uppstår ute hos kund för att kunna korrigera utvecklingsgrenen. Detta kan göras både genom att definiera en löpande uppdatering av alla ändringar från workset B till workset A eller genom att sammantaga (merge) workset A och B.

Ändringshantering

I PCMS är ändringshanteringen tätt integrerad med CM-delen. Detta innebär att filer som ska uppdateras kan relateras till ändringen direkt från ändringsformuläret eller från CM-delen. Relationen är en länk mellan ändringsformuläret och det som påverkas av ändringen. När ändringen sedan införs refereras ändringen från den nya versionen av det objekt som uppdaterats. Ändringsformuläret uppdateras sedan automatiskt med de objekt som påverkas av ändringen (affected) och de uppdaterade revisioner av filerna som ändringen infördes i (in response to).

I och med att den kopplingen finns och att verktyget håller reda på alla beroenden mellan filer, produktstruktur och andra ändringar, gör att en helt ändringsstyrd utvecklingsprocess kan användas, dvs att en ny release definieras som

Föregående release + godkända/införda ändringar => Ny release

Ändringsdokumenten beskriver då skillnaderna mellan två releaser. På så sätt vet man alltid exakt vad som skiljer två releaser från varandra.

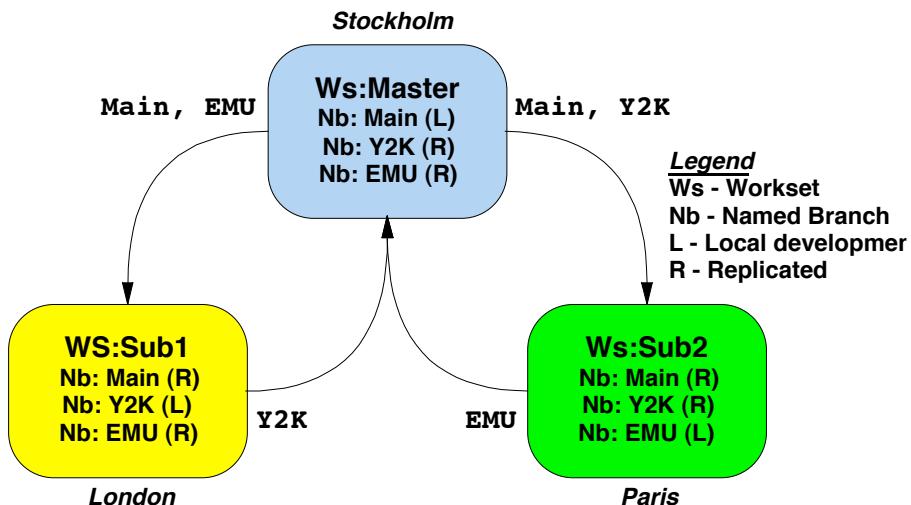
Distribuerad utveckling

De utvecklingsplatser som har PCMS installerat, kan använda sig av Replicator-funktionen för att synkronisera arbetet mellan sig. På en "Master site" definieras vilka "subordinate sites" som ingår i replikering-skonfigurationen.

Där definieras också vilka namngivna utvecklingsgrenar (named branches) som ska replikeras från mastern till övriga siter (en eller flera) och till vilka workset dessa named branches ska importeras. Om det finns utvecklingsgrenar som ägs av sub-siterna och som ska replikeras tillbaka, anges också dessa.

I figur 22 utvecklas huvudgrenen av programmet i Stockholm. Anpassningen av programmet mot år 2000 och EMU har lagts ut på kontoren i London respektive Paris.

Stockholm replikerar huvudgrenen (Main) till London och Paris samt den utvecklingsgrenen av Y2K respektive EMU som hämtats från den andra sub-siten. Arkitekturen innebär att alla siter, via Stockholm, får en fullständig replikering av alla siters grenar.



Figur 22 Exempel på arkitektur för replikering.

En utvecklingsgren kan bara ändras på den site som ursprungligen skapat den. Det innebär att de filer från Main som London ändrar för att lösa år 2000 anpassningen, knyts till den lokala utvecklingsgrenen Y2K. När sedan Y2K har replikerats tillbaka till Stockholm, ser utvecklarna där vilka förändringar som gjorts och kan föra in dessa i Main grenen. Motsvarande metodik gäller för Paris.

Lösningen innebär ett bra stöd för fallet Sammanhållna grupper (avsnitt "Co-located groups"). För utvecklare i samma grupp och som normalt arbetar på samma gren kan det dock vara svårt att arbeta distribuerat (avsnitt "Distributed groups"), men en möjlig lösning är att varje person får en egen gren.

Integrering

PCMS har ett väldefinierat API och kan även integrera med utvecklingsmiljöer som följer Microsoft SCC Interface. För frågor och rapporter mot innehållet i databasen kan browsers eller rapportverktyg med ODBC eller annan koppling användas. Alla tabeller är åtkomliga genom de vyer som beskrivs i dokumentationen.

8.7 PVCS

Översikt

PVCS är ett av de mest spridda SCM-systemen. Det består av ett antal delprodukter varav Version Manager (VM), Configuration Builder (CB) och Tracker är de vanligaste. PVCS förekommer främst på PC-plattform, men finns även för Unix. PVCS har interface till alla större integrerade utvecklingsmiljöer.

PVCS har visst stöd för parallell och distribuerad utveckling, och har många påbyggda SCM-funktioner, men i botten är det ett versionshanter-

ingssystem på filnivå. Det bygger på RCS:s nomenklatur och arkiveringssätt.

- PVCS är bäst lämpat för små och medelstora projekt, där versionshantering av filer eller ändringshantering är det viktigaste. En typisk PVCS-installation har mellan 10 och 20 användare [BW98].
- PVCS är inte lämpat för distribuerad eller parallell utveckling i stor skala, där mergning är ett naturligt förekommande inslag. Dess merge- och synkroniseringsverktyg är inte byggda för det [BW98].

Versionshantering och konfigurationsledning

PVCS VM är filorienterat. Varje arbetsfil har en arkivfil där alla revisioner av arbetsfilen lagras som deltan. PVCS VM är ursprungligen utvecklat för DOS och är lättanvänt. Det senare ovanpålagda Windowsgränssnittet har också utökats med begreppet *projekt*, som tillåter gruppering av filer.

Filarkiven har en *stam* och 0 eller flera *grenar*. Varje sparad exemplar av en arbetsfil kallas *revision*. En gren skapas när en revision, som inte är den sista på stammen, uppdateras. Om man uppdaterar revision 1.2 och revision 1.3 redan finns på stammen, skapas grenen 1.2.1 med revision 1.2.1.0 automatiskt. Antalet grenar och nivåer på grenar är i praktiken obegränsat.

Versioner av system skapas via *versionslablar*, som kan sättas på godtyckliga revisioner i ett projekts filer. Åtkomst till revisioner kan göras med revisionsnummer eller versionslabel. Det finns inget stöd för att följa versionshistorien på ett system.

PVCS VM har en *promotion model*, där filer t ex kopplat till bestämda milstenar får vissa fördefinierade status och därmed begränsas åtkomst för ändringar.

Parallell utveckling och merge

Grenar anges med en utökad siffersekvens. 1.2.1.0, 1.2.1.1 etc är revisioner i en gren som utgick från revision 1.2, och vidare blir 1.2.1.1.0, 1.2.1.1.1 revisioner i en gren på nästa nivå. En användare kan omöjligt vara tvungen att känna till rätt revisionsnummer för sin fil, utan antingen måste man via versionslabel eller *flytande label* identifiera rätt. En flytande label pekar på den sista revisionen på en given gren, t ex grenen 1.2.1. PVCS kan inte grafiskt visa en fils revisionsträd.

PVCS har brister vid storskalig parallell utveckling, där ett merge-verktyg måste kunna klara större konflikter. Merge-verktyget fungerar alltid så att det skapar en resultatfil, som användaren manuellt måste kontrollera före incheckning. Finns det konflikter är de markerade, och filen måste sedan editeras manuellt för att lösa konflikterna. Informationen om själva mergen går förlorad vid incheckning.

Distribuerad utveckling

I PVCS-serien finns PVCS VM SiteSync, som replikerar och synkroniseras arkiv. SiteSync jämför arkiv på olika siter och uppdaterar automatiskt om inga konflikter finns, dvs om en viss ny revision bara finns på

den ena siten. Om samma nya revision skapas på två siter måste den ena lösa konflikten innan synkronisering sker. SiteSync kräver FAT fil-system (dos och windows3.x) och filnamn med max 8+3 tecken.

Ändringshantering

PVCS Tracker är ett lättanvänt och spritt felhanteringssystem, och är inte begränsat till mjukvara. Det kan integreras med PVCS VM så att faktiska kodändringar kan kopplas till rätt Change Request. Ut- och incheckning i PVCS VM kan göras via PVCS Tracker.

8.8 Teamware

Teamware [SMS, SWT99], eller fullständigare Sun Workshop Teamware, är ett CM-verktyg från Sun Microsystems [Sun99]. Det består av verktygen:

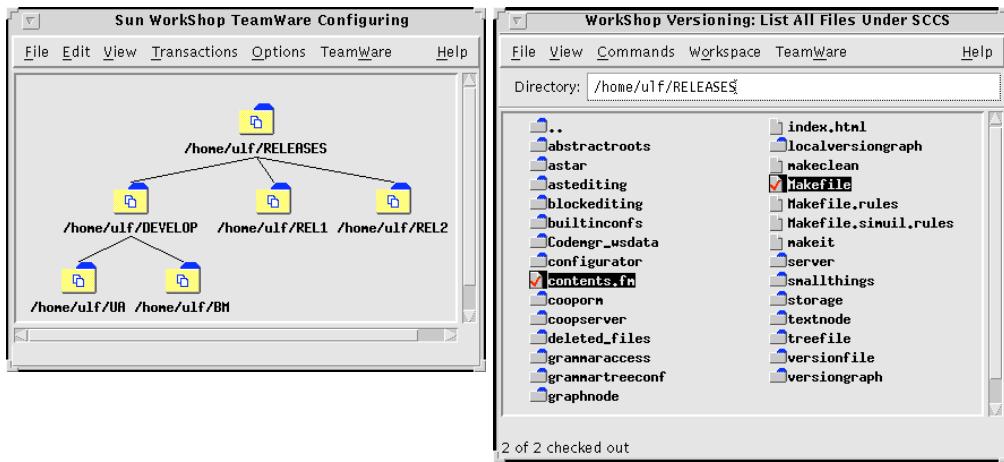
- VerTool - ett grafiskt verktyg för versionshantering.
- CodeMgrTool - hantering av arbetsytor och kataloger.
- FreezePoint - arkivering av konfigurationer.
- PMake - accelerera projektgenerering.
- FileMerge - stöd för 3-vägs merge.

Teamware, eller NSE som det tidigare hette, gjorde modellen Långa transaktioner (avsnitt 4.3) med den optimistiska synen på parallellt arbete utan lösning populär och stöder modellen helt. Stödet innebär framförallt att utvecklingen av hela system görs genom logiska ändringar i relativt stora steg. Hierarkiskt ordnade arbetsytor skapas, uppdateras och synkroniseras med, i Teamware-terminologi, kommandona bringover och put-back.

Parallel utveckling

Figur 23 och 24 visar exempel på användargränssnittet. Figur 23a visar en översiktsbild över hierarkiskt ordnande arbetsytor, där RELEASES är ursprungarsarbetsytan för all utveckling. Det går alltid att skapa nya arbetsytor utgående från de befintliga. En nyskapad arbetsyta är från början en kopia av hela eller delar av sin ursprungliga arbetsyta (sin fader), inklusive hela versionshistorien. Bringover och putback utförs genom drag-and-drop i det grafiska gränssnittet.

En arbetsyta kan innehålla ett helt katalogträd. Figur 23b visar översta nivån av innehållet i en arbetsyta i form av dess kataloger och filer. Inom arbetsytan sker versionshanteringen med SCCS [Roe75], dvs enligt checkout/checkin-modellen (avsnitt 4.1). Utcheckade filer är markerade med en bock. För att få ändra i en fil måste den checkas ut, vilket låser den för ytterligare utcheckning. Vid incheckning skapas en ny version och låset släpps.



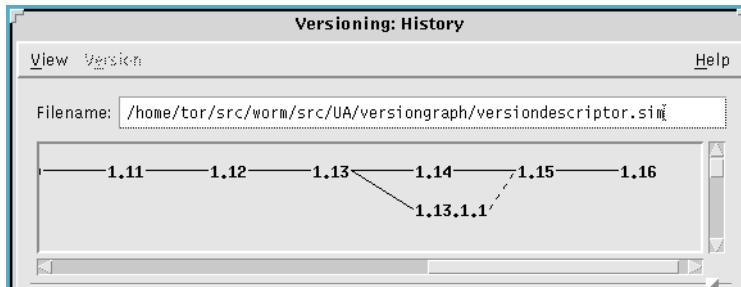
a) hierarkiskt ordnade arbetsytor

b) filer och kataloger i en arbetsyta.
Utcheckade filer är förbockade.

Figur 23 Teamware GUI.

I figur 24 visas versionshistorien för en enskild fil. Grenar skapas normalt inte direkt av användaren, utan är en följd av parallella ändringar av samma fil i olika arbetsytor. När arbetsytorna synkroniseras, via deras gemensamma ursprungsarbetsyta, skapas en gren (version 1.13.1.1 i figuren). Då en gren skapas markeras detta tydligt för användaren och verktyget föreslår att ensning (merge) görs. Om ensningen görs blir resultatet en ny version (1.15). Hanteringen av arbetsytor och ett synkroniseringsscenario beskrivs mer utförligt i avsnitt 4.3, "Long transactions", och då framförallt i figur 5.

Själva ensningen (merge) av filerna görs i Suns verktyg FileMerge. S k 3-vägs merge stöds, vilket innebär att båda versionerna som ska ensas samt deras senast gemensamma version används för att markera ändringarna i respektive gren. Verktyget är radbaserat, dvs alla ändringar identifieras, lagras och presenteras i form av hela rader. Tex när två versioner av en fil jämförs kan varje rad vara oförändrad, ny, borttagen eller ändrad.



Figur 24 Versionshistorien för en fil. Grenar skapas då parallellt utförda ändringar integreras.

Bundna konfigurationer och permanenta grenar

För att versionshantera hela konfigurationer använder man arbetsytor (och alltså inte ”taggning” vilket annars är vanligt). När en hel arbetsyta ska frysas, dvs den aktuella konfigurationen ska arkiveras, skapar man en ny arbetsyta, vilken sedan aldrig ändras. Ett exempel är arbetsytorna REL1 och REL2 i figur 23a. I dessa arbetsytor ska aldrig något arbete utföras, utan de fungerar endast som versioner av hela konfigurationen i RELEASES. Metoden är tids- och utrymmeskrävande då allt kopieras, inkl. alla filers hela historia.

Även permanenta grenar (se avsnitt ”Version control”) hanteras med arbetsytor. En arbetsyta, vilken sedan kan ha ett helt träd av arbetsytor under sig, skapas för att representera grenen. Att senare göra putback tillbaka motsvarar att ensa (göra merge) med den gren varifrån greningen gjordes. Arbetssättet resulterar i en gren-arkitektur med grenar och undergrenar som normalt alltid ensas tillbaka till den gren därifrån de skapades och alltså inte med andra grenar.

Verktyget ”FreezePoint” kan skapa en lista med aktuell version för alla filer i en arbetsyta. Verktyget är dock mer tänkt för att generera checklistor, snarare än något man använder för att enkelt återskapa gamla versioner.

Distribuerad utveckling

Arbetsytorna kan leva sitt eget liv och är tekniskt lätt att placera på olika platser. Man kan även tänka sig att flera utvecklare arbetar parallellt i samma arbetsyta, då enligt checkout/checkin-modellen. Enligt modellen Långa transaktioner bör dock varje enskild utvecklare ha sin egen arbetsyta. Ut- och incheckningen fungerar då som stöd för den egna utvecklingen.

Teamware har ringa stöd för awareness mellan arbetsytor. Snarare stöds en distribuerad situation utan on-line kommunikation, där arbetsytorna är isolerade under längre tider. Synkroniseringen mellan syskonarbetsytor sker via deras fader. En konservativ uppdateringsstrategi stöds eftersom en uppdatering i fadern (putback) inte automatiskt uppdaterar de övriga syskona. Den uppdateringen (bringover) får de själva göra när de ser att fadern uppdaterats och då det passar in i deras egna utveckling. Detta ställer höga krav på awareness så att en utvecklare vet när det finns något nytt att uppdatera med. Krav som verktyget måste stödja i en distribuerad situation.

Två nivåer av awareness och hur man manuellt kan hantera dem i Teamware är:

- *En utvecklare vill kunna se om faderarbetsytan har uppdaterats.* En metod är att helt enkelt prova göra bringover utan att verkligen utföra operationen (“Preview”). Vill verktyget uppdatera arbetsytan är det med nya uppdateringar i fadern.
- *En utvecklare vill kunna se vilka förändringar som gjorts i faderarbetsytan innan han/hon uppdaterar sin egen arbetsyta med ändringarna.* När en arbetsyta skapas måste även ytterligare en arbetsyta skapas, vilken aldrig modifieras utan endast används

som referens. En “preview”-update från fadern till referensarbetssytan visar vilka ändringar som gjorts.

Teamware stöder inte modellen att replikera data. Varje arbetsyta bör endast finnas i ett exemplar. Finns samma arbetsyta replikerad på två platser finns det risk att de uppdateras samtidigt med olika ändringar. En sådan parallell uppdatering kan sedan ge konflikter vid den (automatiska) synkroniseringen av replikerna.

Den modell/process som i stället förespråkas i ”Remote Software Development Using CodeManager” [SWT95] är att använda s k ”bridge workspaces”. I stället för att replikera den riktiga arbetsytan replikeras ”bron”. Uppdateringen av bron sker dock enligt en sekventiell ordning mellan siterna så att de en efter en kan uppdatera sin arbetsyta från bron.

Modellen bygger på att teamware endast stöder ”Single-writer model”, dvs att endast en transaktion i taget får förändra data i en arbetsyta. Broarbetsytorna ses som en virtuell arbetsyta, dvs endast en kan ändra den i taget. (Om en site gör bringover till broarbetsytan så måste alla andra siter uppdatera sig med denna förändring innan någon annan kan göra bringover mot sin broarbetsyta).

Underliggande teknik

Versionhanteringen av filer sker med SCCS. Teamware skapar själv ett SCCS-arkiv i alla kataloger i en arbetsyta. Det tekniska bidraget är att kunna ensa (merge) två SCCS-arkiv.

Att skapa en ny arbetsyta innebär en fullständig kopiering av alla filer och SCCS-kataloger, dvs alla filers hela versionshistoria. Att uppdatera en arbetsyta med bringover och att göra putback innebär en merge av två SCCS-kataloger.

Ett användarkommando mot en arbetsyta, t ex putback, sker som en odelbar operation, en transaktion. Implementationsmässigt delas dock transaktionen upp i flera operationer på varje enskild fil. För en stor transaktion med många filer kan detta ta lång tid, speciellt om användaren som utför operationen sitter på en annan plats och använder ett långsamt nätverk. Resultatet blir att andra användare kan få vänta.

Ingen tidsinformation används för att bestämma huruvida filer har blivit modifierade eller inte. Detta gör systemet robust mot olika tidszoner.

8.9 TRUEchange

TRUEchange (tidigare Aide-de-Camp eller ADC/Pro) från TRUE software är ett CM-verktyg som stöder change-set modellen (avsnitt 4.4), vilken verktyget var först med att introducera. Enligt Susan Dart [Dar97] och Ovum [BW98] är TRUEchange dessutom det enda verktyg som inte är filbaserat utan har en infrastruktur som fullt ut stöder change- sets. Hela produktfamiljen ger ett heltäckande CM-verktyg.

I produktfamiljen ingår:

- TRUEchange - huvudverktyget för CM i produktfamiljen.
- TRUEtrack - tillhörande ändringshanteringssystem.
- TRUErelease - identifierar beroenden och konflikter mellan applikationskomponenter, releasehantering och paketering till distributionsverktyg (det är inte själv ett distributionsverktyg).

TRUEchange är mest intressant för:

- att hantera flödet av ändringar i produkter.
- heterogena utvecklingsmiljöer då många plattformar stöds.

TRUEchange är ett klient/server system skrivet i Java och kan köras på alla plattformar som stöder Javas virtuella maskin. Det grafiska användargränssnittet liknar Windows Utforskare. Verktyget stöder Windows NT, Unix, och VMS-MVS, samt Windows-klienter.

Mer information om TRUE software kan hittas på: <http://www.truesoft.com>.

Verktygets filosofi

Filosofin i TRUEchange är inte att spåra ändringar gjorda mellan olika versioner av en fil. I stället betraktas de logiska ändringar som görs i produkten, och deras påverkan på de filer som produkten består av.

Ett change-set representerar logiska ändringar gjorda till en eller flera filer utgående från en baseline. En change-set definierar syftet med dess uppgift och främjar en förståelse för editeringarnas förhållande till produkten som helhet. Ett change-set kan uppfattas på en högre abstraktionsnivå än en fil.

En styrka med TRUEchange (och change-set modellen) är att ändringarna inte direkt behöver göras i serie. Det är enkelt att "plocka ihop" en ny version av produkten (baseline) med de ändringar man vill ha med, även om de utvecklats vid olika tidpunkter i projektet. I ett versionhanterat verktyg är det besvärligare att plocka ut en viss ändring då det innebär att, från en version, dra ifrån de ändringar gjorda i tidigare versioner som inte ska ingå.

För att kunna utnyttja fördelarna med change-set modellen är det dock viktigt att kunna isolera ändringar. Modellen stöder mest en konservativ utvecklingsstrategi och om inte ändringarna är oberoende kan genereringen av nästa baseline innehålla många konflikter.

Arbetet, vilket givetvis fortfarande är att ändra i filer, utförs i arbetsytor (working area). Vid utcheckning kopplas filen till en change-set och arbetet sker således relativt dess baseline (alla change-sets är relativt en baseline). Vid incheckning måste en hel change-set checkas in samtidigt, då en change-set inte är färdig förrän alla kopplade filer är incheckade. Även andra typer av dokument än källkod kan kopplas till en change-set.

Parallel utveckling

Utvecklare som arbetar i personliga change-sets arbetar i princip alltid parallellt då modellen bygger på att change-sets är oberoende av varandra.

I de fall samma fil ändras i flera change-sets kan konflikter uppstå. Då en konflikt detekteras av verktyget startas automatiskt ett merge-verktyg. 3-vägs merge stöds med presentation av baseline, gren1, gren2 och resultatet i separata fönster, dock utan färg.

Grenar kan också skapas utifrån en baseline. Nya change-sets kopplas till en gren och på så sätt kan en struktur fås på change-sets, även mellan baselines.

Distribuerad utveckling

I princip stöds distribuerad utveckling väl av change-set modellen. En baseline kan replikeras till alla siter där sedan oberoende change-sets implementeras. När nästa baseline ska skapas samlar en site in alla change-sets och genererar den nya versionen. Arbetssättet förutsätter att det går att göra de olika logiska ändringarna oberoende av varandra. Då konflikter ändå uppstår hanteras dessa genom merge som vid vanlig parallell utveckling.

I praktiken blir risken för konflikter stor då all awareness försvinner mellan utvecklarna p g a den geografiska spridningen. Change-sets bör därför distribueras till de olika siterna även mellan baselines för att erbjuda så mycket awareness som möjligt.

I distribuerat "läge" stödjer TRUEchange att en databas skapas med en lagrad baseline. Databasen kopieras till den andra siten. Båda siterna underhåller en lokal gren och en "replikerad" gren. På den lokala grenen sker utveckling som vanligt, medan den "replikerade" grenen underhålls genom uppdateringar från den andra siten.

TRUEsites är ett verktyg som ska möjliggöra att ändringar (i form av change-sets) kan propageras mellan olika siter (eller mellan olika databaser på samma site). Uppdateringar av replikerade grenar kan då göras ofta. TRUEsites är dock fortfarande under utveckling och sådan propagering får i dagsläget därför ske manuellt.

Underliggande teknik

TRUEchange är inte versionsbaserat, dvs alla versioner av en fil sparas inte. Endast basversioner sparas tillsammans med de ändringar som görs i olika change-set. I en change-set fångas de editeringar och relationsändringar i alla de filer som påverkas av ändringen. Endast de ändrade raderna sparas. En ny version av produkten skapas genom att, till den baseline de bygger på, applicera de change-sets som skall ingå. Då skapas nya versioner av alla ingående filer för första gången (alltså aldrig testade). Detta kan innebära att en merge måste utföras om samma filer (och speciellt samma rader) blivit ändrade i olika change-sets.

Susan Dart [Dar97] identifierar tre nyckelaspekter som krävs av ett verktygs infrastruktur för att kunna stödja change-set modellen:

- ändringsidentifiering (change identification) - att identifiera all information relaterad till en ändring, som t ex vem, vad, när, varför och vad som påverkas.
- isolering av en ändring (change isolation) - att fånga endast den information som är relaterad till en ändring så att ändringen är oberoende av andra ändringar.

- inkapsling av en ändring (change encapsulation) - en ändring ska hanteras som ett objekt, dvs med egen identitet. Arbetet är ändringen och detaljnivån på operationer är de för ändringen.

Filbaserade verktyg kan endast erbjuda ändringsidentifiering. Detta kallas ofta "change packages" i de, allt fler, verktyg som stöder det. I [Wib97] diskuteras skillnaden mellan "change sets" och "change packages".

8.10 Visual SourceSafe

Översikt

Microsoft Visual SourceSafe [MSC98] är ett versionshanteringssystem som är bra integrerat med Microsofts utvecklingsverktyg. Till skillnad från många andra CM-verktyg, är det inte filorienterat, utan projektinriktat. Versionshanteringen sker inom ramen av projekt, dvs man använder parallella projekt för att hantera parallell utveckling. Parallelhanteringen på filnivå är rudimentär.

Visual SourceSafe (VSS) är ett enkelt CM-verktyg som saknar många CM-funktioner, men det är användarvänligt och lättanvänt. VSS popularitet växer snabbt. Man kan hitta fler och fler produkter på marknaden som kompletterar VSS med de CM-funktioner som saknas.

VSS finns på Windows plattformar (NT, 95, 98). Det finns också kompatibla versioner på flera Unix plattformar av Mainsoft [MSF98]

VSS är mest intressant för:

- små och medelstora utvecklingsgrupper som utvecklar komponenter och produkter på Windows plattformar, och använder Microsoft utvecklingsverktyg.
- företag som praktisrar outsourcing men inte vill ha någon stark kontroll över utvecklingen av de outsourcade komponenterna.
- företag som börjar införa CM och är mest intresserade av versionshantering.
- företag som vill köpa eller själva bygga produkter som baseras på VSS och som stödjer företags CM-processer.

Versionshantering

VSS versionshanterar filer. Filerna sparas i en databas i en hierarkisk struktur av projekt. Ett projekt motsvarar en katalogstruktur i ett vanligt filsystem. Nya versioner av filer skapas när de checkas ut och in. Parallella grenar av filer finns inte, och det går inte att checka ut och låsa flera versioner av samma fil. Man kan endast checka ut och låsa den senaste versionen.

Parallell utveckling och konfigurationshantering

Om man vill utveckla parallellt, skapar man ett delat (shared) projekt som innehåller samma struktur och filer som det ursprungliga projektet. Alla åtgärder i en fil återspeglas i alla de projekt som filen ingår i. I varje projekt kan man, för alla filer, markera den senaste versionen med ett symboliskt namn (label). Genom att sätta en "pinne" på en fil kan man

välja en äldre version. En pinnad fil inom projektet kan inte ändras. Om en användare gör "get latest" i ett projekt får han den pinnade versionen av filen. Både namnen och pinnarna är synliga bara inom projektet de är definierade i, så andra projekt kan fortsätta skapa nya versioner av filen trots att ett projekt "pinnat" det.

Med delade projekt får man ingen riktig parallell utveckling, utan snarare en möjlighet att konfigurera ihop och fryska versionerna, dvs att göra bundna konfigurationer (baselines). För att åstadkomma en parallell utveckling, kan man förgrena (branch) filer eller hela projekt. Förgrening innebär att en ny kopia av filen skapas, dvs förbindelsen mellan filen och andra projekt som den ingår i bryts. Därefter kommer inte ändringar som görs i filen att överföras till de övriga filerna.

Förgrenade filer behåller alla versioner som finns i det gamla projektet, och dessutom kan man se från vilken version man har börjat i det nya projektet.

VSS har ett merge-program som kan användas för att slå ihop ändringarna gjorda i parallella projekt. I vissa fall görs en automatisk sammanslagning om inga konflikter uppstår.

Distribuerad utveckling

VSS stödjer inte distribuerad utveckling. Om man vill dela vissa strukturer mellan två databaser, måste man göra en snapshot av strukturen och importera den i den andra databasen. Import görs manuellt. Man kan få en lista över de filer som skiljer sig och då kan man checka in dem. Denna typ av kodelning fungerar om man inte har en intensiv parallell utveckling, utan nöjer sig med periodiska uppdateringar.

Även om VSS inte har stöd för distribuerad utveckling, finns det flera andra verktyg byggda ovanpå VSS som angriper problemet. Till exempel, SourceOffSite [SOF98] är en SourceSafe kompatibel klient som fungerar som SourceSafe fast över Internet. TCP/IP protokollet används för kommunikationen.

Integrering med andra verktyg

Integrering med andra verktyg är den starkaste sidan av VSS. VSS är bra integrerat med Microsofts utvecklingsverktyg (som C++, J++, Visual Basic, Front Page, Access).

VSS funktionerna är implementerade som Windows applikationer och som radkommandon. Dessutom finns det ett OLE Automation gränssnitt. Med OLE Automation får utvecklarna programmeringsåtkomst till kommandon och händelser i Visual SourceSafe från Visual Basic- eller Visual C+-baserade program.

Ändringshantering

VSS har ingen stöd för ändringshantering. Men även här finns det flera ändrings- och felhanteringsprodukter som är integrerade med VSS (till exempel Visual Intercept [ELS98] eller PowerTrack [POW98]).

9 Tips vid införande av verktygsstöd

Läsanvisning för de som ska införa stöd för distribuerad utveckling

Tänk igenom hur ni arbetar idag och hur ni vill arbeta efter införandet av distribuerad utveckling. Tänk gärna i termer av de olika fall av distribution som beskrivs i avsnitt 2.1, "Cases of distributed development". De arkitekturer som beskrivs i avsnitt 5, "Architectures" knyter tillbaka till dessa fall, dvs för ett givet fall passar en viss arkitektur bäst. Troligen är många fall aktuella vilket gör att varje enskilt företag själv måste göra övervägande om vilken arkitektur som ska användas. Med fallen klara för sig och en eller två arkitekturer som intressanta kan man sedan selektivt läsa erfarenheterna i avsnitt 6, "Experiences and advice on key areas".

Den terminologi som införs och diskuteras i avsnitt 2-4 är bra att använda vid införandet. Tydliga riktlinjer med en ensad terminologi till både ledning och utvecklare underlättar den fortsatta diskussionen och förbättringsprocessen. Att alla vet att man t ex "använder en konservativ integrationsstrategi med en optimistisk uppdateringsstrategi i en arkitektur med flera servrar vilka automatiskt synkronisera sina rep liker", ger något att diskutera kring.

Kravspecifikation och användningsfall

Både en kravspecifikation och användningsfall bör tas fram. Exempel på krav är:

- Verktyget ska finnas på de plattformar man använder
- Verktyget ska gå att integrera med den övriga utvecklingsmiljön
- Verktyget måste klara kraven på säkerhet
- Det måste gå att implementera stöd för parallell utveckling i den utsträckning och enligt den synkroniseringssmodell (avsnitt 4) som önskas.
- Det måste gå att implementera stöd för önskad ändringshantering, t ex en hierarkisk livscykel.

Det är också bra att definiera ett par användningsfall, scenarier. Exempelvis kan en beskrivning över hur en utvecklare går tillväga för att åtgärda en ändringsbegäran göras.

Låt en grupp jämföra olika verktyg för de definierade användningsfallen. När ett verktyg valts bör en migreringsplan skrivas så att alla inblandade blir informerade och övergången är förankrad. Detta är speciellt viktigt om frågan om vilket CM-verktyg som ska användas har blivit prestigefyllt. Börja med ett pilotprojekt innan verktyget används i full skala. När verktyget/processen införs i stor skala är det mycket viktigt med en grupp testpiloter som kan ingå i nya projekt, dels som entusiaster och dels för att snabbt kunna lösa småproblem.

Erfarenheter

Många företag driver allt mer av sin utveckling distribuerat. Distribu-
tionen kommer smygande och samma metoder, processer och verktyg
används trots att de fungerar sämre och sämre.

När väl nya metoder och ett nytt verktyg ska tas i bruk är det en bra
idé är att tänka igenom hur man vill arbeta distribuerat och eventuellt
komma fram till helt nya metoder och processer. Patcha inte bara den
nuvarande lösningen, utan tänk igenom hur ni verkligen vill arbeta. Det
arbetssätt som används idag är troligen en kompromiss mellan hur ni vill
arbeta och vad som överhuvudtaget är möjligt med nu använda metoder
och verktyg. Begränsningar och regler är kanske dikterade av verktygs-
brister som inte längre finns. Med rätt stöd kan man utnyttja geografiskt
spridda resurser mer medvetet.

Kockums Computer Systems (avsnitt 10.11) har bra erfarenhet av att
stegvis införa ett nytt CM-system, plattform för plattform. Under ett
halvår kördes det nya systemet parallellt med det gamla, innan den
stegvisa övergången inleddes. De var också noggranna med att behålla
populära stödfunktioner från det gamla verktyget, dvs sådana funktioner
implementerades även i det nya systemet.

En stegvis övergång, eller ett ordentligt pilotprojekt (gärna ett lite
mindre projekt men inget ”lätsasprojekt”), är framförallt viktigt i en het-
erogen miljö och då CM-verktyget ska integreras med övrig utveck-
lingsmiljö. Förvånansvärt många praktiska problem av systemkaraktär
kan dyka upp.

Ett alternativ till att köpa ett nytt CM-system är att (vidare)utveckla
ett eget. Detta ger ett tätt samarbete med användarna av systemet, vilket
är en fördel. Dessutom ger det förutom ett användbart system också en
hög CM-medvetenhet hos användarna. En stor nackdel är dock att det tar
lång tid och kostar mycket resurser att utveckla och underhålla egna sys-
tem.

Mycket utbildning, framförallt på det nya systemets filosofi, själ och
synkroniseringsmodell är viktig.

10 Intervjuer

10.1 Inledning

Som nämnts i förordet är intervjuerna personliga. Några officiella uttalanden från företagen har inte eftersträvats. De bör också uppfattas som ”ögonblicksbilder”. CM-området är under ständig utveckling och situationen på företagen förändras kontinuerligt. Alla intervjuade har givits tillfälle att granska och rätta den slutliga texten. Författaren vill än en gång uttrycka sitt stora tack till alla som ställt upp. Utan Er hjälp hade det inte blivit någon rapport.

I intervjureferaten har det intervjuade företagets terminologi använts, vilket innebär att den inte alltid överensstämmer med den terminologi som används i rapporten i övrigt (vilken har försökt hållas så svensk som möjligt).

10.2 Frågeställningar

Detta är exempel på frågeställningar som ledsagat intervjuerna:

Distribuerad utveckling

- Vilka av de fyra fallen av distribuerad utveckling (avsnitt 2.1) har ni?
Hur hanterar ni dem?

Parallel utveckling/awareness

- Hur stor del av utvecklingen sker parallellt av olika utvecklingsgrupper? Hur stor del sker parallellt på olika siter?
- Krävs det mycket kommunikation mellan utvecklare på olika siter, dvs arbetar de tätt tillsammans eller är det snarare som om de levererar färdiga delprodukter till varandra (ungefär som outsourcing eller COTS)?
Sitter utvecklare inom samma projektgrupp på olika siter? Om ja, i vilken utsträckning och varför är inte projekten uppdelade efter hur de sitter?
- Hur synkroniseras utvecklare som arbetar parallellt, dvs hur förhindras att de förstör för varandra.? Vilken synkroniseringsmodell används?
- När *använder* respektive hur blir andra utvecklare *medvetna* om andra utvecklaces ändringar, t ex att en ny version existerar av en fil? Är det någon skillnad på om de sitter på samma site eller inte?
- Kan en utvecklare själv skapa sig en gren att jobba på, eller finns det någon CM-grupp som hanterar det?
- Hur väl känner en utvecklare till:
 - vad andra utvecklare på samma site gör?
 - vad utvecklare på andra siter gör?Vad beror skillnaden på (om någon)? Vilket verktygsstöd används för awareness?

132 Appendix - Tools and Interviews (in Swedish)

- Vad används för verktygsstöd för att göra "merge" på parallella utvecklingsgrenar? Har ni några regler för hur en utvecklare bör gå till väga (t ex att merge först görs till projektgren där test sker innan merge till huvudgren får göras)?

Hantering av ändringar

- Hur koordineras ändringsbegäran? Är det någon skillnad om de hanteras lokalt eller på olika siter?
- Finns det ett centralt arkiv med alla ändringsbegäran eller är arkivet distribuerat på siterna? (eller ett original och kopior på siterna?)
- Hur kommer ni åt status på objekt eller filer på andra siter?
- En buggrättning sker i en produkt (i dess utvecklingsgren), men ska även göras i huvudutvecklingsgrenen. De två grenarna "ägs" av olika siter. Kan situationen uppstå hos er och hur hanterar ni det då?

Inkrementell utveckling

- En fara med inkrementell utveckling är att någon funktionalitet "trillar mellan borden". Hur kommunicerar de olika inkrementansvariga för att minska denna risk?
- Hur arbetar de olika inkrementprojekten i CM-verktyget, i olika grenar? På vilket sätt får senare inkrement reda på att/använder en ny version av tidigare inkrement skapats, t ex p g a av en fejlrättning?

Sekretess och säkerhet

- Har ni krav på sekretess internt eller från era kunder, och i så fall i vilken omfattning? Hur hanterar ni detta?
- På vilket sätt skickar ni data mellan olika siter, med "egen" linje, krypterat på Internet, kurir, ...?
- Har siterna olika säkerhetsklassning, dvs kan en del arbete endast ske på en viss site?
- På vilket sätt påverkar sekretess ert arbete? Hindrar det er från att arbeta så distribuerat som ni annars hade velat?

Arkiv

- Skiljer ni på sättet att arkivera data före och efter release? Om ja, varför?
- Om ni har kopior av arkiv, hur sköter ni synkroniseringen för att hålla dem lika?

Konsistenta utvecklingsmiljöer

- Har ni som krav att hela utvecklingsmiljön ska vara i samma version på alla siter, eller räcker det med att ni vet vilken version som finns på respektive site? Hur hanterar ni problemet?

Införande

- Hur gick ni till väga vid införandet av CM-verktyg? Var det några speciella problem ni undvek/upptäckte?

- Vilka var era krav vid utvärdering av CM-verktyg? Hade ni redan vid utvärderingen en definierad CM-process som verktyget var tvunget att kunna stödja?

10.3 ABB Automation Products

Intervju, Västerås 1998-09-09

Ivica Crnkovic, CM och ansvarig för utvecklingsmiljön
Bertil Emmertz, produktansvarig för Advant Base
Lars Draws, projektledare.

Företag och produkt

ABB Automation Products utvecklar, tillverkar och marknadsför ABB:s automationsprodukter och system för styrning av industriprocesser. Varje år sedan introduktionen av Advant OCS 1993 har ABB varit ledande på marknaden för Open Control Systems (OCS). ABB Automation Products med ca 2000 anställda omsätter 2 700 msek, och som utvecklingsintensiv organisation investerar i F&U med 18%.

Huvudprodukter av ABB Automation Products är Advant- och Satt-produkter. Både Satt- och Advant-systemen är familjer av både hård- och mjukvaruprodukter (ca 50 st stora produkter), vilka tillsammans bygger ett industriellt styrsystem. För systemet görs 10-20 frisläppningar per år med stora nyheter mot kund.

Siter

Bolaget ingår i ABB Automation Segment, Process Control business unit. Både inom bolaget och business unit finns det flera utvecklingscentra: Västerås och Malmö i Sverige, i Tyskland och i USA. Dessutom har man samarbete med flera konsulterbolag placerade i olika städer. Också konsulter som sitter lokalt på plats i Västerås används. Dessutom finns det underleverantörer i Indien och Danmark. Större delen av utvecklingen sker i Västerås där ca 130 personer är engagerade i utvecklingen och i Malmö med 120 utvecklare.

Flera komponenter outsourcas, med ansvaret för både utveckling och underhåll.

Utvecklingen är idag mycket kundfokuserad. Diskussioner med kunder leder fram till s k "roadmaps" vilka är grova planer för produkter och funktioner. Roadmaps leder i sin tur till projekt, t ex ett utvecklingsprojekt. Ett projekt kan i sin tur delas upp i HW, SW, OS, m m.

Relaterade projekt inom olika områden som t ex development, marketing, production och education samordnas och bildar tillsammans ett program.

Funktionell indelning och ansvarsområden

Vid planeringen av nya projekt bör man tänka på vissa viktiga aspekter vilka påverkar sannolikheten för att projekten blir lyckade. Bertil Emmertz gav exempel på ett antal sådana aspekter (i prioritetsordning):

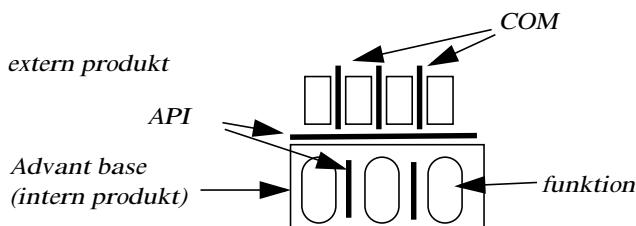
- **Distribution.** Väldigt viktigt att ha en så lokal utveckling som möjligt. Det underlättar möten, framförallt de som behövs med kort varsel (videokonferanser ger inte alls samma resultat). Det är även tydligt att det blir större medvetenhet om vad andra gör, t ex genom kaffepausdiskussioner.
- **Språk.** Erfarenheten visar att en grupp där alla kan prata sitt modersmål har en mer givande diskussion än om de behöver använda annat språk.
- **Tidsskillnad.** Givetvis försvarar en tidsskillnad möjligheten till gemensamma möten. I Västerås måste man t ex vänta till kl 15 innan man kan ha möte där någon från USA ska vara med.
- **Ansvarsområde.** Viktigt att ha klara ansvarsområden så att alla vet exakt vad de ska göra. Om man sprider informationen om de olika ansvarsområdena fungerar den även som "passiv awareness". Den är passiv så till vida att man vet vad andra ska göra, men inte exakt vad de gör just nu.
- **Kulturella skillnader.** En tydlig skillnad är förhållandet mellan utvecklare och chefer, vilket varierar mycket mellan t ex Sverige, USA och Tyskland. Det är därför extra viktigt att ha klara ansvarsområden, vilket gör de olika rollerna tydligare och minskar risken för missförstånd.
- **Funktionell indelning.** Man kan klara av stora delar av uppdelningen mellan siter genom funktionell indelning. Målet är att minska antalet för siterna gemensamma dokument.

Då någon eller några av punkterna inte går att påverka blir de övriga desto viktigare. När det gäller utvecklingen av AdvantBase så har man en distribuerad utveckling mellan olika länder vilka har både tidsskillnad och kulturella skillnader. Kvar att påverka är då ansvarsområden och funktionell indelning.

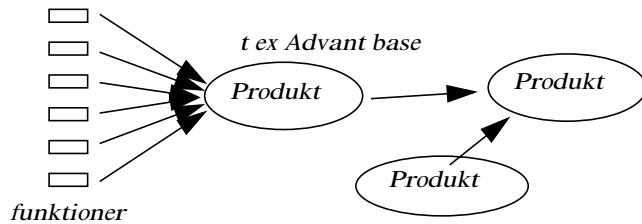
Arkitektur

Tre viktiga arkitekturbeslut har tagits för att underlätta utvecklingen med avseende på bl a distributionen: (1) en gemensam homogen bas för övriga applikationer, (2) funktionell indelning i funktionsmoduler och (3) produktansvariga som integrerar dessa moduler till en produkt.

Advant Base är en intern produkt med syfte att vara en homogen bas för övriga applikationer, se figur 25. Mellan de olika funktionerna i Advant-



Figur 25 Produktarkitektur



Figur 26 Arkitektur för del- och slutprodukter

Base och mellan AdvantBase och de påbyggda applikationerna sker kommunikationen via programmeringsgränssnitt (API). Mellan applikationerna sker det med COM-teknik. Alternativet till AdvantBase är att varje applikation måste hantera en mängd olika funktioner och deras gränssnitt, vilka givetvis då skulle finnas i olika versioner och fungera tillsammans endast i vissa konfigurationer. Det skulle leda till betydligt fler konfigurationer och ett större integrationsproblem.

Det är fortfarande så att konfigurationer med tillsammans fungerande applikationer och AdvantBase måste hanteras, men arkitekturen ger en hierarkisk produktmodell som skalar upp med avseende på antalet moduler och produkter.

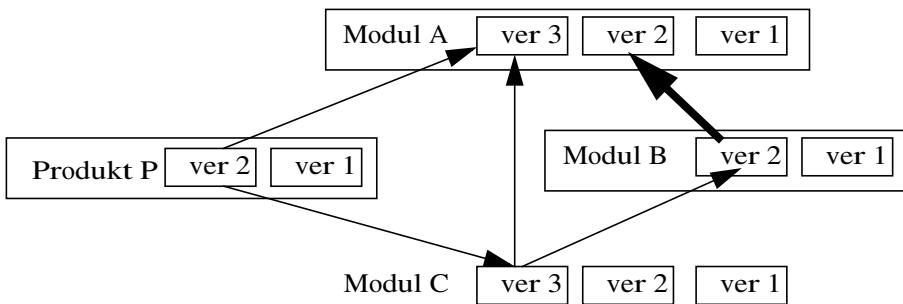
Funktionell indelning innebär att man försöker hålla ett specifikt funktionellt område (t ex utvecklingen av utvecklingsmiljön) inom en site. Tillsammans bildar funktionerna en produkt, vilken antingen kan vara en slutprodukt eller en delprodukt som i sin tur ingår i en annan produkt, se figur 26.

Funktionerna utvecklas parallellt var och en för sig och är uppdelade så att ett så litet beroende fås mellan dem som möjligt.

Vid integration och bygge testas först varje funktion i ett modultest. När modultesterna är färdiga, gör man en bunden konfiguration (baseline) på källkodsfilerna och dokumentation genom att sätta en "tag". En ny version av modulen konfigureras, byggs och paketeras. Den paketerade modulen läggs i en fördefinierad filstruktur på den lokala siten. Modulen är definierad med ett namn och versionsnummer. Motsvarande strukturer finns på andra siter. Produktansvarige och användarna av den frisläppta modulen meddelas. Alternativt bränns en CD eller så skickas den paketerade modulen med mail eller ftp. Inget direkt verktygsstöd används förutom ftp och mail.

Produktansvarig gör produkttest. Före frisläppning görs dessutom en BCT - Big Configuration Test. Då testas hela produkten i en mycket komplicerad miljö och med en så komplicerad konfiguration som möjligt, vilket gör att fel som inte tidigare hittats dyker upp.

Ett potentiellt problem är versionshanteringen av modulerna under modultesterna. För att testa en modul krävs ofta en testmiljö bestående av en eller flera andra moduler. För att behålla konsistens följer man noggrant vilka modulversioner som ingår i produkterna och komponenterna. Varje frisläppt modul innehåller information om vilka versioner av andra komponenter som har använts, vilka kompilatorer och patchar som



Figur 27 Exempel på en inkonsistent integrering

använts, osv. Kontrollen görs både manuellt och med hjälp av egen-skrivna verktyg. Till exempel innehåller varje modul en definitionsfil som används av "make" vid bygge. Den filen refererar till andra moduler och externa komponenter. Vid produktintegreringar kontrolleras, med ett program, om alla moduler refererar till samma versioner av ingående komponenter. Figur 27 visar en inkonsistent integrering, där olika versioner av samma modul används i produkten. I sådana fall skickar verktyget en varning till integratören som kan bedöma om inkosistensen är tillåten eller ej.

Om de använda modulerna utvecklas vidare (kanske som ett resultat av deras modultester) meddelas berörda modulansvariga. Nödvändiga moduler läggs in som bibliotek vid testning.

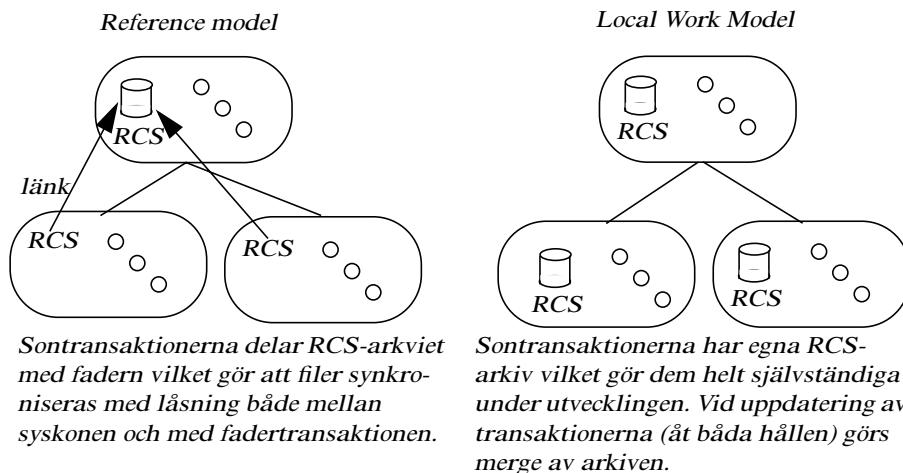
Ett annat problem är att det är svårt komma åt filer på andra siter. Generellt eftersträvas en bättre knytnings mot underleverantörerna, t ex genom att dagligen kunna se hur arbetet går i stället för som nu följa upp en gång per vecka.

Sde och WinSde

Software Development Environment (SDE) är ett egenutvecklat CM-verktyg och utvecklingsmiljö. Det har RCS i botten men ger även stöd för struktur och långa transaktioner. Finns på både Unix och Windows, där det heter WinSDE.

Figur 28 visar två olika sätt att använda SDE. Local Work-modellen ger en funktionalitet lik den i Teamware (avsnitt 8.8) med fristående transaktioner som i princip kan flyttas till olika siter. Referens-modellen ger en möjlighet att skapa sontransaktioner som delar RCS-katalogen med fadern och sina syskon, vilket synkroniseras dem genom låsning på filnivå. Skillnaden mot att arbeta i samma transaktion är att de fortfarande har sina egna utcheckade och genererade filer lokalt.

Vid frisläppning görs en "tag"-ning av alla ingående filer. Status-flaggan, som sätts av utvecklarna, används för att välja ingående filer. Till exempel, status "Stable" markerar de versionerna som är ändrade och uttestade.



Figur 28 Transaktionstyper i Sde

De olika modellerna används vid olika tillfällen:

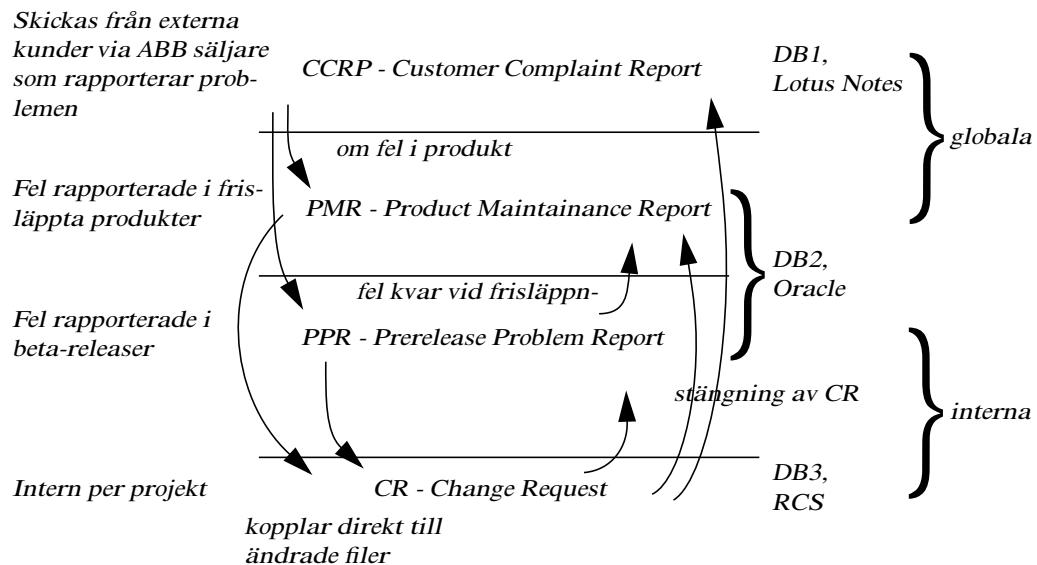
- Alla projektmedlemmar har sina egna arbetsstrukturer skapade enligt referens-modellen. Dessutom görs nya strukturer enligt samma modell när man gör mindre ändringar i form av revisioner av baselines eller frisläppta versioner.
- Grenar av ett projekt, eller parallella projekt skapas enligt Local Work-modellen. En sådan grenning gör man oftast när man börjar utvecklingen av en ny version av en produkt.

Ytterligare verktygsstöd ges i form av ändringshanteringen. Vid checkin refereras till change-request. Ett change-request (CR) är ett dokument som beskriver logiska förändringar (till exempel en felbeskrivning, ett förbättringsförslag eller en ny funktion). I change-request (CR) ser man vilka filer som ändrats. Status på CR visar om ändringen är påbörjad, under pågående ändring, eller avslutat och testad.

Felhanteringssystem

Eftersom ABB Automation Products ger garanti på produkterna i mer än 10 år efter att de inte säljs mer, är felhanteringen oerhört viktigt både som process mot kunderna och som intern hantering. Därför finns det flera verktyg och processer relaterade till felhanteringen (se figur 29).

Kunders klagomål och felbeskrivningar rapporteras i form av "Customer Complaint Report" i en distribuerad databas baserad på Lotus Notes. Om det går att åtgärda felet direkt (till exempel om det handlar om en felleverans), gör man det direkt. De fel som har ursprung i ett systematiskt fel i någon produkt, skickas vidare till PMR (Problem Maintenance Report) system. PMR innehåller alla kända fel i frisläppta produkter. Ett rapporterat fel går genom en felsättningsprocedur, och man kan internt följa upp status av felet (till exempel Investigated, Made, Tested, Verified, Released eller Closed).



Figur 29 Årendehantering och databaser vid felhantering

Vid kod- eller dokumentationrättningen importeras man PMR i utvecklingsmiljön i form av change-request. Alla ändringar inom utvecklings- eller underhållsprojekt styrs i form av change requests. Change requests passerar flera status - från Init, genom Experimental till Closed. När ändringen är avslutad stängs change request och kopieras tillbaka till PMR i form av en ändringsrapport.

Fel hittade i beta-releaser (typiskt vid testningen) hanteras på ett liknande sätt som fel i frisläpta produkter, fast på kortare tid och på ett mindre formellt sätt. Felen rapporteras i PPR-systemet (Prerelease Problem Report). De PPR-er som inte är åtgärdade vid frisläppningen av produkten konverteras till PMR-er.

Hantering av ändringar I projektet "Sirius" har CCB möte i Västerås 2 ggr/vecka. Videokonferens med USA 1 ggr/vecka (vad de vill ha från beroende interface från Västerås).

Arkiv Vid frisläppning går ansvaret formellt över till linjeorganisationen, men ingen fysisk överflyttning av dokument görs.

För att få en säkert reproducerbar målmiljö har man en speciell server där allt byggs. På den servern är endast det nödvändigaste installerat och själva installationen är väl specificerad och dokumenterad.

CM- och utvecklingsmiljö

Ett funktionellt område är utvecklingsmiljön inklusive CM. SDE-gruppen på ca 8 personer (just nu bara 3) har utvecklat och underhåller SDE efter krav och i tätt samarbete med utvecklarna. Arbetsuppgifterna omfattar även regelbundna kurser för anställda och konsulter.

Medvetenheten om CM är stor inom ABB Automation Products, framförallt när det gäller själva hanteringen kring SDE. Förståelsen för ändringshantering och felrapportering är lägre.

Man har nu beslutat att satsa på inköp av ett kommersiellt CM-verktyg istället för att vidareutveckla och underhålla det egna verktyg. Det pågår därför just nu en utvärdering av olika CM verktyg (ClearCase och SourceSafe), vilken också SDE-gruppen ansvarar för.

10.4 ABB Robotics Products

Intervju, 1998-09-23

Stefan Forssander, chef för avdelningen metoder och produkter med ansvar för att driva (frisläppa nya produktutgåvor av styrsystemet) och förbättra utvecklingsprocess och verktyg.

Företag och produkt

Utvecklingen sker distribuerat med ca. 100 utvecklare i Sverige, 20 i Norge, 5 i Frankrike och 5 i USA. Bolaget i Norge äger en del av mjukvaran (ett s k ämnesområde) och fungerar i praktiken som ett delprojekt i Västerås utgåveprojekt. Arbetsformen blir mest som Outsourcing. Både Frankrike och USA fungerar i huvudsak som underleverantörer till ämnesområden i Västerås och används främst av kompetens- och/eller resursskäl.

Produkten är ett robotstyrsystem, S4, som ingår i ABB Robotics alla robotar, samt tillhörande optioner och online/offline-verktyg i PC-miljö. Ca. 3 fullständiga utgåvor släpps per år tillsammans med 5-10 revisioner på tidigare utgåvor. Varje utgåva innehåller numera flera produkter för olika plattformar. Totalt vidareutvecklas och underhålls ca. 2.5 miljoner rader källkod.

Utveckling och integration

Mellan två utgåvor driver man ett s k utgåveprojekt, vilket är ett paraplyprojekt med uppgift att samordna alla aktiviteter/projekt som utvecklar funktioner som skall ges ut i en gemensam ny styrsystemutgåva. Mjukvaran är uppdelad i olika ämnesområden, vilka utvecklas av olika projektgrupper med egna ämnesområdesansvariga.

Utvecklingen är centrerad kring integrationsbaselines (testbäddar). Nya testbäddar byggs och släpps på beställning, eller minst två gånger i veckan, under integrationstester. Efter det att en testbädd skapats hanteras alla ändringar m h a av formella SPR:er (System Progress Reports) på CCB-mötens. Spårbarhet fås genom att i SPR:er ange till vilken utgåva de hör. Dessutom loggas alla rättningar för att exakt kunna se vad som levereras till en viss testbädd. På så sätt vet de alltid vilka ändringar som gjorts både till en utgåva och till de olika testbäddarna.

CCB-mötens hålls som riktiga möten en gång i veckan, där Norge är med m h a konferenstelefon.

När det gäller leveranser från Norge hanteras dessa som COTS. Norge äger som sagt en del av systemet, nämligen den mjukvara som hanterar det som är specifikt för målningsrobotar. De levererar hela denna del,

men mycket kommunikation uppstår i samband med integration, vilken görs i Västerås. De har alltså i viss utsträckning distribuerad utveckling, men en helt centraliserad integration.

Som CM-verktyg används RCS. Det enda arkivet (RCS repository) finns i Västerås och de andra siterna använder remote checkout/checkin. Kommandona är av bolaget egenutvecklade UNIX-script som använder e-mail för att föra över filer från Västerås till Norge. De replikerar även alla originalbibliotek i sin helhet till Norge varje natt, vilket gör att de har tillgång till samma miljö som Västerås, men med ett dygns förskjutning. För att få snabbare byggtider replikeras även objektkod och andra genererade filer mellan siterna.

Inga grenar används i RCS, utan arbetet sker istället parallellt i flera kompletta utgåvebibliotek. Ämnesområdena fungerar som permanenta delprojekt i den totala utgåveprocessen och ämnesområdena förser därför samtliga pågående utgåvor med resurser. Samma personer jobbar ofta med flera olika utgåvor parallellt, men alltid inom de objekt som deras ämnesområden förvaltar. Utvecklarna levererar in sina filer till utgåvorna m h a UNIX-scripten. Utgåvorna administreras av utgåveintegratorer och ändringar inom ämnesområdena (delprojekten) hanteras av projektintegratorer, som skapar nya baselines med jämn mellanrum. Arbetssättet blir i princip lika för alla siter eftersom alla originalbibliotek ligger i Västerås.

Som stöd för awareness mellan utvecklare och siter används ett intranet åtkomligt för alla siter. Där publiceras nya planer och annan information, t ex alla SPR:er. All information som hanteras via webben fungerar bra för alla siter.

Inkrementell utveckling

En inkrementell utvecklingsmodell används. Ofta används flera interna integrationer innan den slutliga produkten görs. Kunden kan få en tidig ofullständig prototyp för utvärdering. För att förhindra att funktionalitet ”trillar mellan borden” används en matrisorganisation med ämnesområdesansvariga och funktionsansvariga som tillsammans bevakar helheten. Ämnesområden äger objekt. Funktionsansvariga bevakar samspelet mellan samtliga inblandade objekt genom flera ämnesområden. Dessutom finns alltid en integrationsansvarig som bygger ihop varje inkrement.

Ändringshantering

Företaget har ett väl fungerande SPR-system (System Progress Report), där samtliga nya funktioner, förbättringsförslag samt felrapporter registreras. I botten ligger en Oracle-databas med ett web-baserat interface. Den är central (finns bara en), men ligger på intranet och är åtkomligt från alla siter.

Eftersom all integration är centraliserad och de inte tillåter att några andra siter än Västerås ger ut egna produkter finns det ingen risk för bugrättningar endast i en produkt. Ämnesområdena hanterar, som tidigare sagts, samtliga pågående utgåvor och inför felrättningar i alla berörda utgåvor parallellt.

Sekretess

Alla siters utveckling nät ligger bakom brandväggar. Data skickas mellan siterna på egen krypterad lina till Norge och på band med DHL till Frankrike och USA.

Sekretesskravet med kryptering försvårar dataöverföringar. Frankrike tillåter ej kryptering, så det går t ex inte att använda samma koncept i Frankrike som de kör mot Norge.

Ett krav på varje site är att brandvägg finns. Om den saknas får de ej hantera vissa koddelar.

Arkiv och utvecklingsmiljö

En fullständig kopia på dubbla band görs i samband med frisläpp. Ingen "tag"-ning sker så banden är samtidigt säkerhet och arkiv, dvs sättet att få tillbaka tidigare utgåvor då dessa inte längre finns online.

Företaget har dessutom en central rutin för backuper. Förvaltningen av dessa kopior sköts av gemensamma resurser för datordrift.

Målet är att ha samma versioner av hela utvecklingsmiljön på alla sitter. Inga speciella verktyg används för att i efterhand se vilken version av miljön som använts vid en speciell byggnng, utan endast RCS används. Däremot tas backup av hela miljön då den ändras.

10.5 Combitech Software

Intervju, Jönköping 1998-09-17

Intervjuade: Anders Olofsson CM-ansvarig, Bengt Rydh projektledare, Kent Eriksson delprojektledare.

Företag och produkt

Combitech Software är ett konsultföretag som arbetar med uppdrag inom området tekniska realtidssystem. Förutom rena konsultuppdrag åtar sig Combitech Software även totaluppdrag som inbegriper utveckling av både programvara och maskinvara åt kund. Combitech Software har idag ca. 136 anställda konsulter.

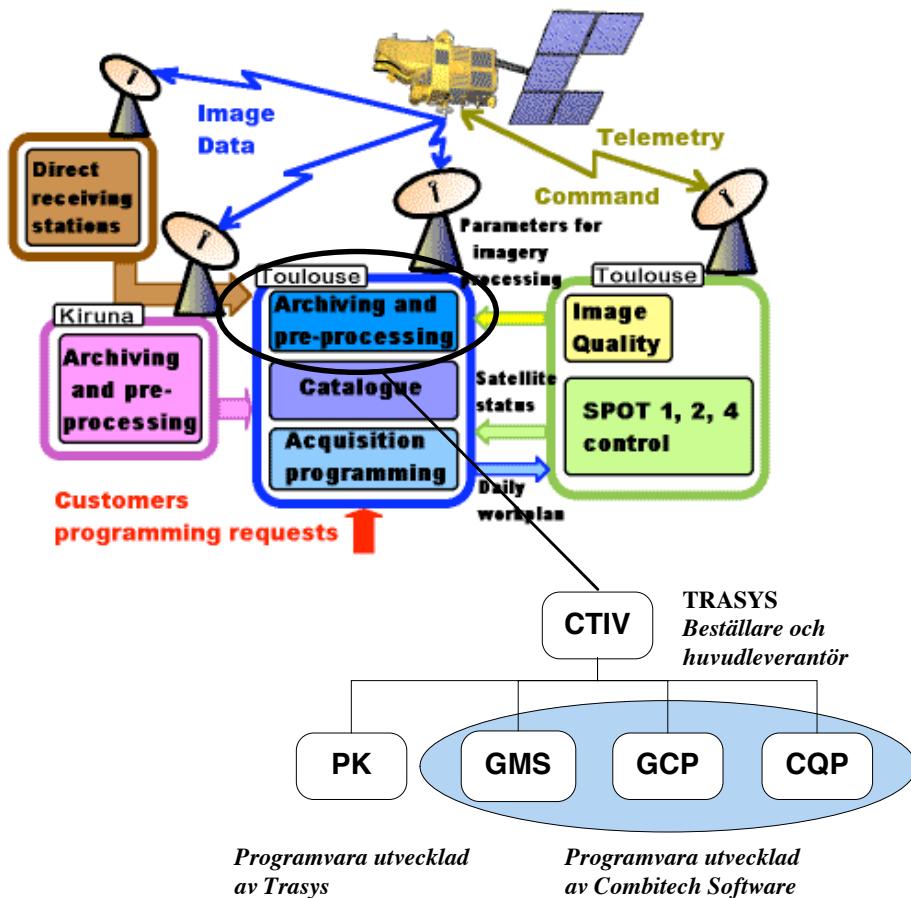
Projektet

CTIV-projektet ingår som en del i det större SystèmVegetation-projektet som i sin tur är en del av SPOT4-systemet, se figur 30.

SPOT 4 sköts upp i mars 1998 och syftet är att studera vegetationens utbredning över jorden. Sponsorer för projektet är bl a EU, Belgien, Sverige, Frankrike. Slutkund är franska rymdstyrelsen, Centre National d'Etudes Spatiales, CNES.

Projektet startade 1996 och Combitech Software (vidare kallat CS) övertog ansvaret för utveckling av programvaran från den ursprungliga leverantören hösten 1997. Hittills har CS lagt ca. 14 manår i projektet. De har 1 års garantiåtagande på produkten och 10 års underhållsansvar. Produkten godkändes av CNES vid en Site Acceptance Test 981201.

Karakteristiskt för detta projekt är rymdindustrins höga krav på kvalitet, som i det här fallet definieras av CNES. CNES som beställare



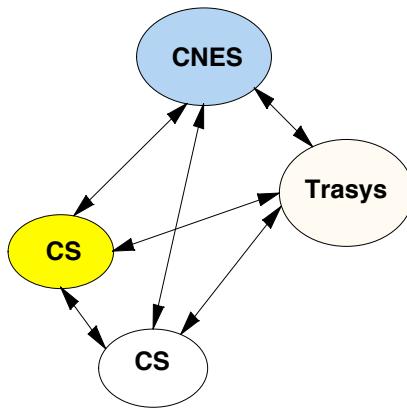
Figur 30 SPOT 4

har en hög teknisk kompetens och deltar aktivt i arkitektur och design av systemet, ända ner på kodnivå.

Konfigurationsledning

CS ”ärvtde” rutiner och verktyg för konfigurationsledning från den ursprungliga leverantören. Det har inneburit att CS i många fall har fått gilla läget och istället arbeta med att förbättra existerande rutiner så långt som möjligt. Ingående komponenter är:

- Konfigurationsledning: Beskrivs i CM-planen samt detaljeras i egna interna instruktioner vid behov.
- CM-verktyg: RCS med vissa påbygganader. Används endast för kod.
- Ändringshantering: Manuell på papper (pärmsystem). Access-databas för registrering och uppföljning av ändringar.
- Dokumenthantering: Manuellt i filsystemet på gemensam nätverksdisk med manuell versionshantering.
- Egenutvecklade kommandofiler för hjälp vid releaser



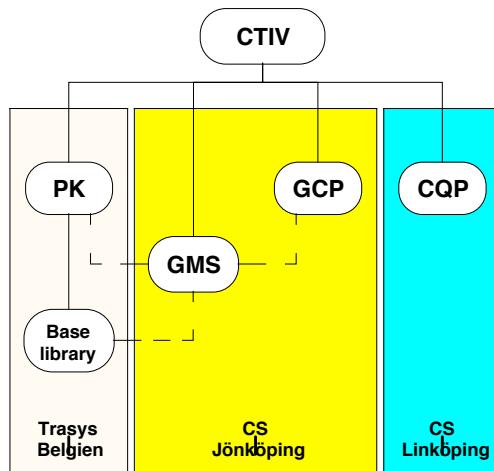
Figur 31 Utvecklingsplatser och distributionsvägar

Distribuerad utveckling

Slutkund och beställare är CNES i Frankrike. Utvecklingen av CTIV sker hos Trasys i Belgien och hos Combitech Software i Sverige (i Jönköping och Linköping). Programvaror, dokument, fax och e-mail skickas regelbundet mellan dessa platser. Det som skiljer sig från det normala i detta fall är att programvara och dokumentation levereras direkt till slutkunden (CNES) för tidiga verifieringar av funktionalitet och innehåll.

Produkten

Figur 32 visar en enkel produktstruktur och vem som utvecklar de olika delarna av CTIV. Processing Kernel, PK, utvecklas av Trasys i Belgien. GeoModelling Server, GMS, Ground Point Control, GCP och Contrôle Qualité des Produits, CQP utvecklas av Combitech Software i Sverige. De streckade pilarna i figuren visar beroenden mellan delprodukterna och hur både CS och Trasys använder varandras delar.



Figur 32 CTIV Produktstruktur

Inom CS sker utvecklingen på två siter. I Jönköping utvecklas GMS och GCP och i Linköping CQP. Två nivåer av distribution kan urskiljas, dels den mellan CNES, Trasys och CS samt den som sker internt inom CS (se figur 33).

Varje delprodukt som utvecklas av CS lagras i ett gemensamt RCS-arkiv som ligger på en unix-server. PC- och Unix-maskinerna ligger på ett gemensamt nätverk där alla diskar är åtkomliga från respektive miljö. Alla programreleaser sker från detta RCS-arkiv. Det som distribueras till CNES går också via Trasys ftp-server, eller i vissa fall skickas det på CD. Alla informeras genom distribution av faxkopior på skickade delivery notes.

Hantering av ändringar

En rad olika dokument används för att hantera olika typer av ändringar inom projektet:

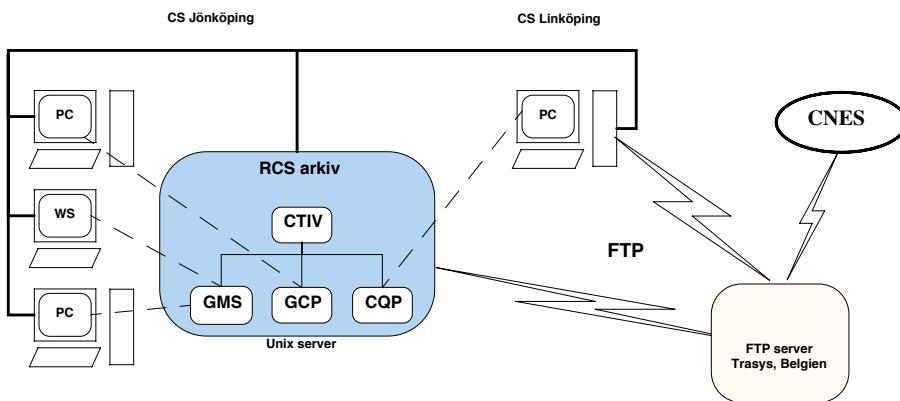
- Anomaly Sheet, AS: används när en avvikelse gentemot kontrakt eller specifikation upptäcks i konfigurationsstyrda program- eller maskinvara
- Software Modification Sheet, SMS: används för att dokumentera ändringar i programvaran. Där står exakt vilka filer och versioner som har ändrats för att korrigera en AS. Korsreferenser finns mellan AS och SMS.
- ICR, Implementation Change Request: begäran om ändringar i gränssnitt
- Demande de Modification, DM: begäran från kunden om kravändringar. Den utvärderas av CS ifråga om tid och pengar.
- Demande de Derogation, DR: begäran av leverantören om avvikelse från krav
- Proposition de Modification, PM: används av leverantören vid begäran om ändringar gentemot ursprungskontrakten. Dessa är väldigt formella och byråkratiska att hantera.

De ändringar som utvecklarna oftast kommer i kontakt med är AS och SMS.

Anomalier kan rapporteras av alla parter. Att stänga en öppnad AS följer en formell procedur. CS får endast stänga de AS de själva öppnat och endast om det sker före mötet som sker vid varje key-point. Övriga AS stängs först av CS och sedan slutligen av Trasys.

CS och Trasys har var sin anomali-databas. Fördelen är lättåtkomligheten lokalt men det är problem att hålla databaserna synkroniserade. T ex kan anomalier som stängs på en site vara fortsatt öppna på den andra siten.

Att placera ett fel eller anomali kan vara ett problem i en distribuerad miljö. Mellan företagen ligger ansvaret för utplaceringen hos Trasys. Inom CS är det relativt lätt att placera till rätt delprodukt. Klara ansvarsområden med tekniskt ansvariga gör att rätt person oftast får uppgiften att korrigera problemet.



Figur 33 Nätverksarkitektur

Synkronisering

De beroenden som finns mellan de olika delarna i produkten och mellan företagen gör att kraven på synkronisering är höga. Det leder till att kommunikationen är mer formell. Signerade dokument måste faxas för att få med de underskrifter som krävs. Det leder ibland till en del praktiska problem som krånglade faxar och svårslästa utskrifter. Eftersom alla fax sparar i pärmar går dessa inte att integrera med övriga elektroniska dokument. Kommunikation sker även via e-mail (endast informell kommunikation) och vid vanliga möten. Vid akuta ärenden ordnas telefonmöte under dagen. Månadsmöte (vid samma bord) hålls regelbundet med Trasys, då erfarenheten säger att man missar för mycket med endast telefonomtöte.

Olika möten används för att synkronisera projektstatus:

- Veckomöten för CS projektgrupp. Bl a går man igenom status för alla anomalier och "actions" (mindre formella anomalier).
- Månadsmöte med Trasys och CNES.
- Key-point möten. Genomförs i slutet av varje utvecklingsfas för att konfirmera statusen på produkten vad gäller funktionalitet, kvalitet och ändringar.

Key-points

Utveckling och integration av programvara styrs med hjälp av key-points. En key-point är ett basunderlag (baseline) som fastställs efter en utvecklingsfas. För varje key-point finns ett antal leverabler definierade. Detta kan vara dokument och/eller kod med en viss status. Vid key-point mötet går man också igenom rapporterade, genomförda och utestående ändringar och beslutar om hantering av dessa.

Följande key-points mellan Trasys och CS har använts under utvecklingen:

- Architectural design
- Detailed design
- Pre validation

- Factory Acceptance

Konsistenta utvecklingsmiljöer

Den miljö som används av CS vid utveckling måste stämma överens med den miljö där Trasys bygger, testar och använder programvaran. Utvecklingsmiljön finns noggrant beskriven vad gäller versioner på operativsystem, kompilatorer och andra 3:e parts verktyg som används. Uppgraderingar initieras från Trasys och sedan har CS ca. två veckor på sig att uppgradera sin utvecklingsmiljö.

Kravet från CNES är att den senaste versionen av verktyg som ingår i utvecklingsmiljön alltid ska användas. Detta går dock inte alltid att uppnå på grund av inkompatibilitet mellan verktygen, men det är dit man strävar.

Dokumentreleaser

Vid leverans av formella dokument skrivs en Document Delivery Note som i detalj beskriver de dokument som levereras. Den innehåller titel på dokumenten, registreringsnummer, filnamn, version och var filerna finns att hämta på Trasys FTP-server. Dokumentfilerna kopieras till Trasys FTP-server och delivery noten samt påskrivna förstasidor av varje dokument faxas till både Trasys och CNES.

Programreleaser

Varje release är identifierad i RCS med hjälp av en "tag". Den "taggen" används för att kopiera ut de ingående källkodsfilerna till operativsystemet. Ett script används sedan för att jämföra releasen mot föregående release och påvisa skillnaderna. Dessa skillnader ifråga om filnamn och revisioner måste motsvara de ändringar som är dokumenterade i de Software Modification Sheets (SMS) som beskriver rättningarna som gjorts i releasen. Dessa SMS:r refereras sedan i den Software Delivery Note som åtföljer varje leverans.

Källkoden kopieras sedan till Trasys FTP-server. Releasenoten skrivs under och faxas tillsammans med påskrivna SMS:er till Trasys och CNES. Om dokument ingår i leveransen följer dessa procedurer som beskrivs under Dokumentreleaser.

Om samma ändring görs i flera grenar, t ex både i en buggfix-gren och i huvudgrenen är de med i var sin SMS, men med "länk" mellan varandra.

Till Trasys levereras ingen binärkod. I stället levereras endast källkod tillsammans med noggranna instruktioner för hur systemet ska byggas. För CQP:n börjar installationsbeskrivningen av systemet med hur operativsystemet ska installeras på en från början tom maskin och fortsätter vidare till en komplett installation. Allt ska vara i rätt version och installeras i rätt ordning.

Mottagning av dokument och programvara

Pappersdokument arkiveras i pärmar och dokument som levereras via ftp lagras i en katalogstruktur på den gemensamma projektdisketten. Utvecklingsverktyg som levereras till CS registreras i ett dokument som beskriver utvecklingsmiljön. Där står namn, version, mottagningsdatum, installationsdatum och från vilken release verktyget ska börja användas.

Tillhörande manualer lagras i pappersarkivet. Alla projektdeltagare informeras om mottagna leveranser via e-mail.

Erfarenheter

Alla principer som gäller konfigurationsledning vid lokal utveckling gäller dubbelt upp eller mer vid distribuerad utveckling. Antalet leveranser ökar kraftigt, både från och till den egna utvecklingsmiljön. Om detta dessutom ska utföras manuellt eller med enkla verktyg måste en större arbetsinsats beräknas. Vid en stor release kan 4 personer vara syselsatta under en hel dag bara med att verifiera innehållet och korrigera Delivery Notes. I ett verktyg som stödjer principerna konfigurationsledning fullt ut, behöver man i stort sett bara trycka på knappen för att få ut en korrekt release som är fullt dokumenterad.

Sammanfattning:

- Planera noggrant! Gör en detaljerad CM-plan som bl a beskriver ansvarsområden och process för ändringar, namnkonventioner för alla ingående objekt, hantering av utvecklingsmiljön, releaserutiner m m.
- Satsa på ett heltäckande CM-verktyg. Vid större projekt kommer kostnaden för detta snabbt att tjänas in.
- Sträva efter att samtliga inom ett distribuerat projekt använder samma databas för ändringshanteringen. Detta ger en bra översikt.
- Dokumentera ofta förekommande rutiner noggrant så att någon annan än du kan utföra dem.
- Använd intranet för distribution av projektgemensam information. Det är svårt att få folk att läsa de papper som skickas ut.

Mer information om CTIV

Mer information om projektet kan du finna på nedanstående adresser:

- <http://www-projet.cst.cnes.fr:8050/>
- http://spot4.cnes.fr/spot4_gb/index.htm
- <http://www.vgt.vito.be>

10.6 Enator

Intervju

1998-10-27. Kristian Dristig, Ansvarig för EES' telecom designcenter, Enator Elektroniksystem AB

1998-11-18. Mats Fredriksson, projektledare "Gränslös utveckling", Enator System Design AB

Företaget

Enator har 5000 anställda på 60 orter i fem länder. Enator Teknik, som ingår i Enator, har 300 konsulter på 5 orter i Sverige: Göteborg, Västerås, Karlstad, Uppsala och Kista. De är självständiga bolag men de arbetar alla med utveckling inom elektronik-, programvara- och IT-området.

I två projekt, Pablo och Porträtt, har man utnyttjat resurser från flera bolag. Man har i så hög utsträckning som möjligt flyttat personer för att kunna utveckla lokalt. En anledningen är insikten att projekt fungerar mycket i korridorerna där den informella kommunikationen är viktig.

För att kunna få en väl fungerande distribuerad utveckling måste man utveckla en bättre modell för informationsutbyte. I ett internt projekt inom Enator Teknik som kallas "Gränslös utveckling" arbetar man med att ta fram praktiska tips och metoder för distribuerad utveckling. Bland de viktigaste framgångsfaktorerna för en distribuerad verksamhet är en lyckad mänsklig kommunikation.

Pablo

I ett projekt, Pablo, mot Ericsson, behövde man utnyttja resurser från flera orter. I detta fallet undvek man den distribuerade situationen så mycket som möjligt och de inblandade fick tillfälligt flytta till Göteborg. Positiva erfarenheter från projektet är:

- Företagets goda kunskap om Ericssons arbetsätt och utvecklings- och målmiljö gjorde det möjligt att på kort tid skapa en likvärdig miljö innehållande nödvändiga verktyg samt en egen målmiljö.
- Länk till Ericsson (ERINET) med vilken man levererar via Ericssons vanliga leveransverktyg.
- Genom att samla ihop alla inom ett projekt så slipper man "linjebrus", vilket möjliggör hög effektivitet med ca 36 timmar/vecka i projektet.

Nackdelen är att personer måste veckopendla under hela projektet. För längre projekt är detta sällan möjligt.

För att upprätthålla ett eget målsystem krävs kontinuerlig kommunikation med och kunskap om kunden. Allt eftersom kunden vidarutvecklar målsystemet måste projektet uppdatera sitt eget. En praktiskt väl fungerande lösning för att kunna hantera flera versioner av målsystemet är att använda sig av olika uppsättningar av hårddisk, en uppsättning för varje version av målsystemet. Då räcker det med en dator per maskintyp (Sun, HP, etc) så byter man lätt mellan olika versioner av målsystemet.

Porträtt

I ett senare projekt, Porträtt, är situationen mer distribuerad. Dels är det fler som arbetar via fjärrinloggning, dessutom är fler parter inblandade. Förutom Enator är även Ericsson mer aktiv i utvecklingen både genom anställda och andra konsulter.

Huvuddelen av utvecklingen görs lokalt i den för projektet uppsatta miljön (liknande den för Pablo-projektet). Då flera olika företag är inblandade, kontraktsmässigt i flera led, är fjärrinloggningen mer problematisk när det gäller sekretess och åtkomstregler (endast Enator-anställda kan fjärrinlogga direkt mot projektmiljön). Genom att möjliggöra inloggning även från Ericssons miljö kan deras identifieringsrutiner användas. Ericsson-anställda och övriga konsulter använder då ett system, RACOM (Remote Access Communication), vilket genom privata lösenordsgeneratorer möjliggör identifiering vid inloggning via modem. Väl identifierad i

Ericsson-miljön kan den identiteten få rättigheter även i projektmiljön. Själva åtkomstskyddet implementeras med grupper i Unix.

Alla inom projektet arbetar mot samma ClearCase-server. Leveranser till Ericsson sker genom att göra merge till en leveransgren vilken, med hjälp av MultiSite, replikeras till en ClearCase-server på Ericsson. Förtom Porträtt hanterar Ericssons server flera andra relaterade projekt. För varje projekt görs en egen MultiSite-replikering från projektserver till den centrala servern.

Gränslös utveckling

Enator Teknik bedriver ett projekt kallat "Gränslös utveckling" inom vilket man utreder möjligheterna att utnyttja distribuerad utveckling. Som konsult har man ofta projekt där arbetet är distribuerat mellan "hemmabasen" och kunden, men det förekommer även geografisk spridning inom företaget. De enskilda bolagen vill normalt inte vara större enheter än ca 40 personer men man vill ändå tillsammans kunna ta stora uppdrag. Genom resursutnyttjande över bolags- och ortsgränser vill man kunna effektivisera, ta större uppdrag, och efter behov kraftsamla specialkompetens.

Projektet är avslutat så till vida att det har lett fram till ett arbetssätt, "Gränslös utveckling", vilket nu används gentemot kund. Resultatet är dock levande och förändringar görs i takt med nya erfarenheter. Som en del i det framtagna arbetssättet används sedan tidigare mer generellt framtagna metoder vilka samlats i en "verktygslåda". Framförallt metoderna för "Praktisk Projektstyrning" och "CM" är relevanta för distribuerad utveckling. Man har också tagit fram en "kokbok" med praktiska "råd & dåd". De är mycket praktiska och talar om hur man ska göra i konkreta fall. De ligger i web-format på Intranet och har bl a länkar in i verktygslådan.

I arbetet har man hittills inriktat sig på: mänsklig kommunikation (med enkel teknik) och fysiska aspekter (som nätverk).

Mänsklig kommunikation Alla måste känna (för) samma sak. Om man har olika mål och ambitionsnivåer blir det lätt missförstånd. Det är viktigt att man lär känna varandra först genom face-to-face möten innan man arbetar distribuerat. Dvs vanlig gruppbildning är viktig.

Även för längre projekt där en stor del av arbetet ska göras "hemma" bör man i början sitta hos kund för att lära känna varandra. Första gången man arbetar för ett företag är det dessutom viktigt att lära sig företagskulturen (och landets kultur då det är aktuellt) för just den kunden.

Fysiska aspekter Viktigt med bra kommunikationslänkar och hög bandbredd. Som mötesstöd är de stora stationära videokonferensrummen inte så flexibla. Däremot bra erfarenhet av enklare mötesverktyg kopplade till den egna datorn. En webkamera på datorn som gör att man åtminstone kan se stillbilder av varandra (inte alltid uppdateringen är så snabb) tillsammans med telefon och verktygsstöd för att dela fönster ger en bra mötesmiljö. "Delningen" av fönster innebär t ex att alla kan se ett fönster vilket en manipulerar. Finns även modellen att alla kan ändra i

ett dokument som alla kan se, då enligt någon typ av "floor control". Inte en så tekniskt avancerad lösning men det fungerar bra för att t ex diskutera kring kravdokument, kod och liknande.

När det gäller CM så är det svårt att hitta några generella tips. Som konsult är det oftast kunden som styr val av verktyg, CM-process, etc. Normalt är det fördelaktigt om alla utvecklar mot "samma filer". En central server eller flera servrar som synkroniseras fungerar båda bra. Det viktiga är modellen med samma filsystem/filstruktur och att man undviker att ta lokala kopior.

Inom företaget används nyhetsgrupper flitigt för att bilda virtuella grupper inom vilka mer specialiserade diskussioner förs.

En ny teknik som håller på att utvärderas är VPN (Virtuella Privata Nät). Det är en teknik för att skapa subnät som om de var galvaniskt åtskilda trots att befintliga nät utnyttjas. Det ska göra det möjligt att skapa privata åtskilda nät även genom brandväggar.

10.7 Ericsson Microwave Systems AB

Intervju, Göteborg 1998-09-11

Annita Persson, Business Unit CM Manager

Lennart Larsson, ClearCase ansvarig för bolaget

Viktor Ohlsson, Business Unit CM Manager

Företaget

Ericsson Microwave Systems (EMW) affärsidé bygger på att företaget utgör ett högteknologiskt centrum inom Ericsson för mikrovågsteknik, antennteknik och avancerad signalbehandling. I företagets kärnenheter Mikrovågsteknik och Höghastighetselektronik, Innovationscenter, Mikrovågsdesign och Antenner utvecklas deras kärnkomplicer. Denna teknikbas utnyttjas inom Ericsson Microwave Systems i två affärsenheter: Försvarselektronik och Mikrovågskommunikation.

Ericsson Microwave Systems är en internationell ledare av digitala mikrovågs radio-länkar och radio-basstationer för mobiltelefoni. I samarbete med andra Ericsson bolag utvecklar de nästa generation av GSM:WCDMA. De konstruerar, utvecklar och producerar även avancerade aktiva antennsystem för mobila nätverk.

Företagets huvuddel inom försvarsområdet är radar och kommunikationssystem, där de utvecklar, konstruerar och producerar avancerad elektronisk utrustning för armén, flottan och flygvapnet.

EMW har cirka 4 700 anställda varav ungefär 75 procent är tekniker. Huvudkontoret ligger i Mölndal men verksamhet finns också i Borås, Linköping, Kista, Skövde och Lysekil.

Verktyg, metoder och processer

Bolaget använder flera olika CM-verktyg, bl a ClearCase och ExcoConf. Framför allt används ExcoConf i de något äldre projektet, där man sedan en lång tid tillbaks har använt verktyget. Rekommendationen på bolaget är att alla nya projekt skall använda ClearCase för att hantera program-

varan. ClearCase används inom ett antal enheter på EMW, men inte på samma sätt överallt.

I ClearCase-miljön finns det 82 licenser, 300 vyer (ca 1/person), ca 65 VOB:ar i det öppna nätverket och 5 VOB:ar i det hemliga nätet. Ca 50 GB data hanteras där ca hälften är vyerna, dvs arbetsytorna för varje vy inklusive deras genererade filer.

För alla applikationer som bolaget använder utses en person som är applikationsstödsansvarig (ASA), dvs någon som kan mer än den vanlige användaren om verktyget, ansvarar för att utbildningar anordnas för användare, installerar upgraderingar och tar first-line supporten internt på bolaget. ASA för ClearCase är Lennart Larsson.

Förutom ClearCase så används en koncerndatabas, GASK2, och ett koncerngemensamt register, PRIM. GASK2 fungerar som ett gemensamt originalarkiv för hela koncernen med åtkomstmöjlighet i hela världen. Arkivet och registret tillsammans används även för synkronisering och kommunikation mellan projekt och olika siter. För att få snabbare åtkomst till senaste projektinformationen, använder bolaget ett verktyg, LOCARC, som abonnerar på ändrad information från registret, PRIM, samt cachar den nya informationen, dvs delar av GASK2, lokalt på en server.

PRIM (PRoduct Information Management) är koncernens centrala register för arkivering, abonnemang, kopiering och distribution av dokument. Det är även här man hämtar nya revisionsnummer samt håller ordning på vilket produktutförande som består av vilka revisioner. Som stöd för awareness kan man prenumerera på ändringar iregistret, t ex att ett nytt revisionsläge skapats för ett speciellt dokument eller på nya ändringsbeskrivningar.

I ClearCase hanteras revisionsläge genom att en label sätts. För att få en tydligare koppling mellan revisionslägen och ändringsbeskrivningarna i PRIM och ClearCase, planerar bolaget en automatisering mellan dessa två verktyg. I dagsläget används prenumerationfunktionen mest för nya versioner av hårdvaran, medan awareness för mjukvara helt hanteras inom ClearCase. Det är dock viktigt att revisionslägen i ClearCase även kommer in i PRIM, så att produktutförandet blir fullständigt. T ex fungerar felrapporteringen endast om läget är registrerat i PRIM.

LOCARC (Local Archive) är ett lokalt kopiearkiv för att förbättra prestanda och navigationsmöjligheter för dokument lagrade i GASK2. Dokument i GASK2 kopieras, översätts till pdf-format och lagras lokalt, vilket möjliggör åtkomst via webb-läsare. Uppdatering på förändringar av dokument från GASK2 sker varje natt. Verktyget används framför allt för verksamhetsstyrande dokument men även för ett fåtal projekt.

GASK2 (Generellt Arkiv System för Kommunikation 2) är koncernens gemensamma originalarkiv för frysta dokument och produkter.

Distribuerad utveckling

Fjärrarbetssplatser för Mölndal finns både i Öckerö och i Varberg. Dessa använder egna fast uppkopplade krypterade linjer, vilket ger samma miljö som om de satt lokalt, men längsammare prestandamässigt.

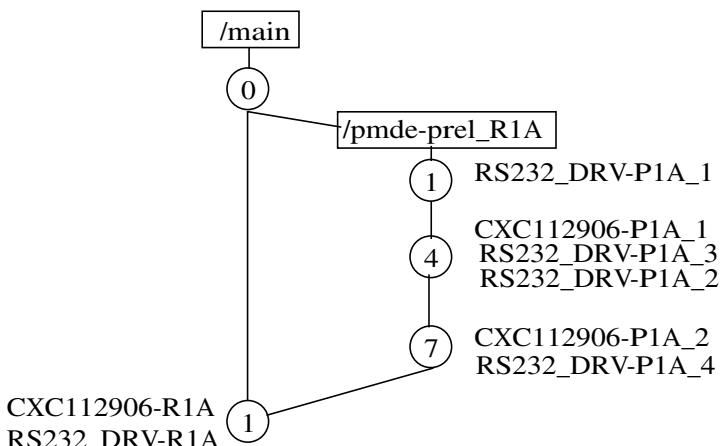
I det projekt i vilket Viktor Ohlsson är Configuration Manager utvecklas en mobilsimulator där både hård- och mjukvara tas fram. Utvecklingen i projektet är distribuerat med fyra huvudsiter i Kista, Mölndal, Nürnberg och Finland. Dessutom är ytterligare två Ericssonföretag och ett företag i Halmstad underleverantörer.

Göteborg fungerar som designcenter till Kista. Gävle tillverkar hårdvaran (kort) enligt beskrivningen i GASK2 och Kista bygger ihop. Mjukvaran utvecklas i Mölndal och Nürnberg. Både källkod och "laddningsmoduler" leveras till Kista. Själva leveransen sker genom att en label sätts i ClearCase. Kista får reda på detta antingen direkt via t ex email eller genom prenumerationen i GASK2. De har sin egen VOB, synkroniserad med MultiSite, i vilken de kan läsa de grenar där den frisläppta produkten ligger. Kompilering sker både på respektive utvecklingsställe och i Kista.

Parallel utveckling Grundregeln är att ha så lite parallel utveckling som möjligt på filnivå, men olika typer av projekt kräver olika arbetsätt. Arbetsättet skiljer sig mellan nyutveckling och underhåll, eller som det också kallas "utgrenade projekt".

Nyutveckling Direkt under en VOB ligger alla produkter och underprodukter i en flat struktur. Varje produkt och underprodukt är så stor att en person kan få överblick. En person (eller eventuellt en liten grupp) är ansvarig för ett antal produkter eller underprodukter. Varje sådan produkt eller underprodukt motsvaras av en katalogstruktur.

För filernas versionsträd gäller att nyutveckling sker i arbetsgrenar, vilka döps enligt regeln projektnamn plus det revisionsläge man siktar på (t ex pmde-prel_R1A), se figur 34. Under projektets gång kan man släppa



Figur 34 En fils versionshistoria med grenar och lablar

```

element * CXC...P1A_3 -nocheckout      ← versionsnummer för modul. Ökas när
# element * RS232...P1A_7 nocheckout   ← en senare version önskas/finns.

.....
# Do not change below this line!
element * CHECKEDOUT
element * /main/pmde-prel_R1A/LATEST
element * /main/0 -mbranch pmde-prel R1A ← vid utvecklingen använder utvecklaren
                                              senaste utgåvan av modulen, dvs den
                                              här raden är bortkommenterad och
                                              raderna nedanför används i stället.

LATEST innebär att man ser de senaste
incheckningarna på grenen.

```

Figur 35 Konfigurationsspecifikation

preliminärutföranden. Detta görs genom att sätta en label med en produktidentitet och ett versionsnummer på den aktuella grenen. Exempelvis är RS232_DRV-P1A_1 och CXC112906-P1A_1 (laddningsmodulens label på ingående filer) båda det första preliminärutförandet till revisionsläge R1A. Projektet avslutas med att dess gren merges (ensas) till huvudgrenen och en label med revisionsläge sätts (t ex CXC112906-R1A).

Normalt ändrar man aldrig något i någon annans moduler. Om det skulle bli nödvändigt skapar man sig en temporär gren i vilken ändringarna görs. Genom email blir den ansvarige för modulen medveten om ändringarna och kan, genom merge, föra in ändringarna till huvudgrenen/projektgrenen.

För att underlätta utvecklarnas hantering av konfigurationsspecifikationer skapar man vid projektstart en specifikation för projektet som definierar en partiellt bunden konfiguration vilken alla utvecklare utgår från, se figur 35. Specifikationen hanteras därefter av användarna själva. Den är förberedd för vissa ändringar med bortkommenterade rader som enkelt kan bytas med andra som då i stället kommenteras bort. Den vanligaste ändringen är att öka versionsnumret på en label för att på så sätt få med en nyare version av ett annat projekts filer.

Underhåll En gren skapas för varje ny felrapport (på alla inblandade VOB:ar). Felrättningslevereras alltid med nya revisionslägen, dvs aldrig som patchar. En person som arbetar med en felrapport kan behöva modifiera i ett antal delprodukter, dvs här rättar man i delprodukterna oavsett vem som ansvarar för dem. Detta innebär också att flera personer kan rätta flera felrapporter samtidigt (en gren per felrapport).

Multi Site Huvudregeln är att låta varje site vara ansvarig för så stora block som möjligt, dvs i första hand hela VOB:ar. När ansvaret för en VOB ändå måste delas sker separeringen i första hand av hela kataloger och i sista hand ändrar man i samma katalog.

Tre exempel på fall av distribution med MultiSite:

- En del VOB:ar replikeras på många Ericsson siter, men endast en site har skrivrättigheter per VOB, dvs en skriver och många läser.

- EMW och en konsult delar några VOB:ar. De arbetar parallellt på båda siterna inom samma VOB och gör sina ändringar i egna grenar, en för EMW (t ex emw_prel_R1A) och en för konsulten (xxx_prel_R1A).

Ett exempel på problem som kan uppstå i det här fallet, är att en fil saknas på båda grenarna, varpå båda siterna upptäcker detta och skapar filen. Resultatet blir två (olika) filer i olika grenar av katalogen med samma filnamn. För att undvika problemet, delas ansvaret upp mellan siterna. En site ansvarar för vissa kataloger och den andra siten för resten.

- En VOB delas mellan tre siter. Även kataloger delas och alla tre vill kunna skapa filer i samma kataloger. Varje site har en egen arbetsgren för katalogerna. Där läggs egna filer. MultiSite ser till så att alla grenar uppdateras på alla siter. Ett cron-jobb (dvs ett kommando/script som körs, ofta regelbundet, vid en viss tidpunkt) gör sedan själva ensningen (mergen) av grenarna.

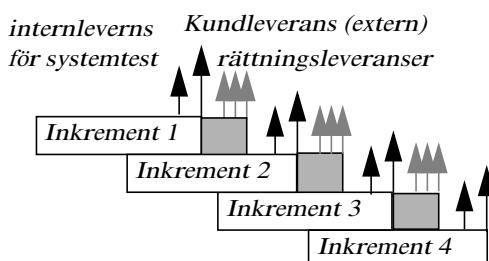
Hantering av ändringar Man skiljer på ändring av krav och på felhantering. Ändringar av kraven hanteras av ett centralt CCB som även beslutar om frisläppning. I stora projekt kan det även finnas lägre nivåer.

Felrapporter kommer in till en person som överlämnar dem till en s k TR-akut (Trouble Report), som är en grupp med representanter från varje site och som oftast träffas via telefonmöte. De placeras felet på rätt site där en handläggare tar över felrapporten och ser vad som behöver göras.

På varje site finns ett system, MHS (Modification Handling System) med en MHO (Modification Handling Office) och en handläggare definierad för att se vilka som ska vara mottagare för felrapporten. I systemet finns även produktansvariga som får felrapporter till sig. Systemet finns utvecklat i IBM med kopplingar till GASK2 och PRIM.

Inkrementell utveckling Utvecklingen är uppdelad i 8 huvudfaser/inkrement. För varje inkrement sker först en intern leverans för systemtest och sedan extern leverans till kund. I de fall någon kund rapporterar in fel rättas dessa och rättningsleveranser sker, se figur 36. Leverans av inkrement sker genom att sätta en label på den gren inkrementet utvecklas. För att frisläppa en hel produkt sätts en label på alla ingående filer.

För varje inkrement finns en dokumentplan som anger när och vem som har ansvaret för rättningsleveranser eller tillägg.



Figur 36 Inkrementell utveckling

För projektledning använder EMW PROPS som är en TollGate-modell applicerad på en projektstyrningsmodell. Vid utvecklingen, konstruktionen, använder bolaget en nyligen framtagen utvecklingsmodell, SPEED (System Product EnginEering and Development). SPEED ska fungera som ett generellt ramverk, utifrån vilket olika projekt kan skapa olika applikationer genom att efter behov komplettera ramverket med specifika detaljer. SPEED baseras huvudsakligen på: kunskapstillväxt, inkrementell utveckling och integrationsdriven utveckling.

Sekretess och säkerhet Grundregeln är att man litar på sina anställda och skyddar sig mot ofrivilligt felaktigt användande och utomstående. Inom projekten har alla tillgång till alla filer medan man separerar rättigheter mellan projekten. Det är mycket för att skydda de anställda själva som onödig information inte lämnas ut.

När det gäller icke försvarshemliga projekt så kan krypterad linje användas från fjärrarbetsplatser. Det finns också bärbara datorer med stöd för kryptering som kan användas vid hemarbete.

För hantering av försvarshemlig information, hanteras detta på ett separat nätverk med ett särskild inloggningssörfarande.

Konsistenta utvecklingsmiljöer För att få samma utvecklingsmiljö framförallt inom projekten används typavändarmiljöer, vilket finns både på Unix och på PC (där den heter ESOE - Ericsson Standard Office Environment). Alla applikationer startas via miljön vilket gör det både säkert och lätt att underhålla.

Inköpta och egenutvecklade verktyg hanteras i en gemensam databas, där alla har minst en person utsedd för att vara applikationsstödsansvarig. Dessutom klassas varje programvara för att ge ett stöd i val av verktyg vid uppstart av nya projekt. Fyra nivåer finns för en (version av en) applikation: (1) rekommenderad (alla nya projekt ska använda), (2) begränsad användning (de projekt som pågår kan fortsätta att använda), (3) under avveckling (beslut om datum för stopp av användning finns samt en avvecklingsplan) och (4) makuleras.

Verktygsbyte Bolaget gjorde hösten 1993 en utvärdering av olika CM-verktyg. ClearCase valdes med huvudargumentet att det fanns på alla de plattformar som krävdes. Systemet var i drift mars 1994 och under 4 år har det endast varit en halv dags partiell driftstörning (en plattform utslagen), då en patch råkade installeras under fel identitet.

PDM (Product Data Management) Bolaget har inget datoriserat verktyg för hantering av PDM. Användningen av Ericssons beskrivningssätt av produkter och dess tillhörande dokument är i sig ett PDM-system, ett system som hanteras manuellt.

Inom WCDMA-projektet håller man på att bygga på en projektanpassad modul baserad på verktyget Metaphase, för att hantera produkter, produktstrukturer med tillhörande dokumentation på ett för projektet gemensamt sätt. Systemet beräknas kunna tas i drift tidigt 1999.

10.8 Ericsson Mobile Communications

Intervju, Lund 1998

Alf Ek, CM-ansvarig.

Företaget

Ericsson Mobile Communications AB (ECS) tillverkar mobiltelefoner för GSM och andra system. Mjukvaruutvecklingen sker numera på ett tiotal olika siter i världen, från att från början helt och hållet varit i Lund. Produkterna levereras i miljontals exemplar. Kännetecknande för utvecklingen är att många projekt löper parallellt, med för branschen kort utvecklingstid och kort livstid.

Utvecklingsprocessen

Sedan starten av utveckling av programvara för GSM-telefoner, har i Lund en källkodsbas byggts upp. Alla produkter (typiskt telefonmodeller) har hämtat och hämtar sin programvara därifrån. Flera produkter är samtidigt i olika stadier av nyutveckling, dvs incheckad kod för en produkt får genomslag i andra produkter direkt. Produkter på marknaden uppdateras också kontinuerligt. Gemensamt för alla produkter är att de delar källkod i största möjliga utsträckning.

Gradvis har ett behov uppstått att andra siter behöver göra egna GSM-relaterade produkter och använder då Lunds kodbas. Eftersom time-to-market är en överlevnadsfaktor i branschen, har inte arbetssätt och mjukvara kunnat anpassas innan utvecklingen började ske på andra siter. Utvecklare utanför Lund har alltså nödsakats betraktas på samma sätt som utvecklare i Lund, dvs de kan ligga on-line och arbeta mot Lunds kodbas. Med stora delar gemensam kod medför detta störningar, eftersom antalet utvecklare ökat dramatiskt de senaste åren. Dessutom är det mycket opraktiskt och långsamt för distansarbetarna, svårt för dem att testa alla produkter de påverkat. Kvalitén på det s k huvudspåret blir ofta instabil och rentav obygbar.

Detta gör att produkter ibland tidigt väljer att lämna huvudspåret, för att kunna ha kontroll över ändringar. På lång sikt betyder detta att produkterna blir kraftigt divergerade och fejlrättningar måste göras på onödigt många olika ställen.

Dessa praktiska problem har givit ett arbetssätt som innebär att vissa siter tar snapshot av koden, och utvecklar lokalt tills deras kod är klar. Ambitionen är att deras kod sedan merges in i Lunds kodbas, men det är svårt att säkerställa att alla rättnings verklig förs in i andra produkter. På samma sätt är det svårt för dem att uppdatera sig med vad som händer i Lunds kodbas.

Numera har ECS en uttalad policy som endast låter Lunds personal checka in filer i Lunds kodbas. Kod som utvecklats på annan site, måste godkännas av Lund innan den förs in i kodbasen.

Under 1998/1999 genomförs en förändring av mjukvaran och av arbetssättet för att stödja den distribuerade utvecklingen, samtidigt som ett nytt CM-verktyg införs.

Nuläge

Sekretess/säkerhet

ECS använder inte CM-verktyg för sekretess. De anser att det ger för lite jämfört med den administration det medför. Alla som har skrivrättighet till kodbasen har automatiskt skrivrättighet på alla filer. Med nuvarande arbetssätt blir problemet att de inte kan hantera individer på andra siter, vilket medför att de ger grupper på andra siter läs eller läs/skrivrättighet. Det är upp till varje site att populera sina grupper, och individerna godkänns på siten, inte i Lund.

Ändringshantering

Ändringshantering är svårt att synkronisera. Felrapporter på mjukvara i telefoner utvecklade på en site hamnar lätt bara på den siten, fast samma fel mycket väl kan finnas någon annanstans också. I Lund finns en feldatabas hanterad av mjukvaruavdelningen i Lund, som gäller produkter som använt Lunds källkod. Varje mjukvaruprojektledare, i Lund och på andra siter, håller sig dessutom ofta med egen fellista, som gäller just den produkten och som "lever" tills produkten lämnats över till underhåll (sw maintenance). Dessa fellistor tenderar att inte alltid offentliggöras för andra produkter.

Varianthantering

Varianter hanteras som egna produkter enligt ett master-slave förhållande, vilket innebär att när mastern släpps, släpps också dess slavar. Dessa slavar innehåller då ett fåtal andra filer än mastern. Det vanliga exemplet är att telefoner har olika språkuppsättningar, men i övrigt samma mjukvara.

Parallel utveckling

Parallel utveckling förekommer inte med verktygsstöd. Behöver flera personer ändra i samma exemplar av en fil samtidigt, checkas filen ut av en och de övriga arbetar på egna ej utcheckade filer. Ändringarna mervas sedan ihop manuellt.

Datalagring

Ordinär backup-tagning. Varje site sköter sitt. Inget får utvecklas på egen hårddisk, allt på nätet. På hårddisk endast program och s k derived objects.

Releasehantering och distribution

Releaser styrs av produktledningen och av marknaden. Vad som ingår mjukvarumässigt i första releasen av en produkt är oftast en kompromiss mellan lämplig lanseringstidpunkt och färdig funktionalitet. Vidare planerade mjukvaruuppdateringar i produkten görs ca två år framåt, med felrättningar och ibland ny funktionalitet.

Krav på distribuerad utveckling

Program måste utvecklas på olika platser i världen. Detta pga att olika kompetenser finns på olika platser, att vissa delar med fördel utvecklas

nära sin marknad, att det inte i Lund finns möjligheter att expandera tillräckligt för att möta den exploderande marknadens krav.

Programvaran måste delas upp i komponenter. En komponent ägs, tillverkas, underhålls, ändringshanteras på en site. Utveckling av komponenter drivs av egen organisation (t ex p g a krav från hårdvara) och av produktledning (t ex p g a funktionalitet). Viss komponentutveckling läggs ut på underleverantörer (typiskt ett konsultbolag). Skillnaden mot intern komponentutveckling blir att underleverantörerna helt och hållet själva CM-hanterar sin programvara, medan de inom företaget kommer att ha insikt i varandras utveckling, utan att för den skull kunna störa varandra.

En produkt tas fram på en site. Produkten beställer och använder komponenter från olika siter. Dessa komponenter måste passa ihop. Ändringshantering på en komponent kommer att begäras från olika siter. Ett globalt CCB kommer att behövas i någon omfattning.

Sekretess

En komponent ägs på en site och CM-verktyget ska förhindra andra siter från att ändra i komponentens huvudspår. Teoretiskt sett ska allt kunna distribueras mellan olika servrar inom företaget. Distribution sker på eget nätverk.

Inkrementell utveckling

CM-verktyget ska stödja inkrementell utveckling, så att sådan kan utföras isolerat, utan att störa andra, samtidigt som utvecklingen enkelt kan flyttas från en site till en annan,

Ändringshantering

Ändringshanteringen måste följa noggranna rutiner. Fel upptäckta i en produkt kommer att resultera i begäran om uppdatering av någon komponent, på någon site. Dessa ärenden måste gå att följa och status på varje ärende måste vara offentlig. Change Board (CCB) måste inse vad det innebär att ändra i snitten.

Parallel utveckling/Awareness

CM-verktyget ska stödja parallel utveckling, inom en site eller mellan två siter, så att sådan enkelt kan mergas. Vad som sker på olika siter måste synkroniseras och offentliggöras.

Datalagring

Inga nya krav. Varje site sköter sitt. Läsrättighet hos varandra.

Releasehantering och distribution

Releaser dokumenterar vilka versioner av vilka komponenter som ingår. Hur komponenter är kompatibla med varandra måste offentliggöras. Än en gång krävs en stark och förståndig produktledning som kan inse betydelsen av detta.

10.9 Ericsson Telecom

Intervju

98-10-26. Gunnar B. Nilsson, projektledare för Orion-projektet

98-10-29. Stefan Törnqvist, projektledare för PU/Access inom Polaris-projektet.

Företag och projekt

Orion-projektet omfattar ca 340000 mantimmar och 227 miljoner kronor. Den totala projektorganisationen består överst av linjeorganisationer, exempelvis Business Solution (BS, systemägare och de som samlar ihop produkter och bygger den totala lösningen i form av en AXE-växel) och Business Management (kontakten mot marknaden). Under linjerna finns produktenheter (PU) vilka utvecklar de fristående produkter som systemet byggs ihop av. Exempel på PU:ar är Residential Service (utvecklar tjänster riktade mot privatpersoner), Business Communication (företagstjänster), Network Service Control (trafikhantering) och Access (abonmentutrustning).

Det finns 8 st olika PU varav 7 används i Orion-projektet: 1 i Stockholm, 2 på Irland, 1 i Finland, 1 i Danmark, 1 i Holland och 1 i Italien. Varje PU är i sin tur indelad i Lokala Design Centra (LDC) inom vilka själva kodningen och produktutvecklingen sker.

Totalt innebär detta att utvecklingen för en produkt kan ske distribuerat på ca 15-17 olika platser.

Integrering, test och frisläppning

Hela frisläppningsprocessen följer en strikt fastställd tidsplan och all programproduktion görs centralt. Alla PU har samma tidsplan och vet när kodningen ska vara klar och levereras till BS (Business Solution). Leverans görs med hjälp av ett system som heter Swaxe (IBM stordatormiljö). BS bygger därefter centralt ihop en s k testdump och lägger på stationsdata. Allt ska byggas och laddas in på en station och fungera så pass bra att ett s k "basic call" går att genomföra. Detta tar ca en vecka för första dumpen. I praktiken är det alltid någon PU som inte levererar allt i tid vilket då fördröjer hela projektet. När ett basic call går att utföra distribuerar BS ut testdumpen till alla PU och LDC. Varje LDC får på så sätt sin testmiljö med identisk dump och stationsdata som övriga LDC (vilket för Orion-projektet innebär ca 30 st stationer). Efter 2-3 veckor med testing och debugging görs en ny leverans till BS. Hela förloppet itereras ca 3 gånger innan systemet slutligen helhetstestas av BS och släpps till kund.

Som fördelar med centraliserad programproduktion nämns:

- Att alla har samma miljö på alla plattformar. Om fel hittas i någon annans produkt är det lätt för ägaren att själv återskapa felet.
- För PU fungerar det bra att lägga upp en plan som följer totalprojektets tidplan och sedan hålla sig till denna.

Inget ytterligare CM-verktyg används internt för versionshantering.

En speciell testorganisation utför själva testningen i respektive PU. Testledare leder testningen och felrapporter (Trouble Report, TR) skrivs på de fel som identifieras. Testledaren måste ha kunskap om hela systemet för att kunna identifiera och placera alla felrapporter utifrån en given felsituation. Även om felet upptäcks för en viss funktion skrivs sällan felrapporter på funktioner utan i stället direkt på olika moduler/filer. En felrapport (ett fel) delas därför upp i flera felrapporter (en per fil) som då hör ihop. Stöd för att hålla ihop dem logiskt finns genom att i felrapportformuläret kunna ange relaterade felrapporter (t ex de från samma funktionella fel).

Under själva felrättningen får de ansvariga för respektive fil/modul informellt ta kontakt och prata med varandra för att bestämma var och hur felet ska åtgärdas. Det är sedan viktigt att alla relaterade felrättningar samtidigt läggs in i Swaxe, dvs under de s k AD-dagarna då man levererar in till Swaxe. Synkroniseringen för detta sker helt manuellt via telefon ("Är du klar?"), vilket kan resultera i mycket kommunikation de sista dagarna.

Leverans

Frisläppningar görs ca var 6:e månad. Vid varje leverans levereras alltid en basutgåva av systemet, en s k GAS (General Application System), men det görs även flera leveranser av kundanpassade system, s k MAS (Market Application System). Numera sker leverans av både GAS och MAS ungefär samtidigt. Förr kunde en MAS släppas upp till sex månader efter leverans av GAS.

Till skillnad från GAS, som följer processen beskriven ovan, så testas en MAS närmare kund. Det är svårt att få upp någon relevant trafik i en teststation så endast en första test sker där innan leverans sker till kund (där sluttest görs).

Organisationen har en "Small Company Approach" där varje PU har eget vinstintresse. Det innebär att de själva säljer sina idéer direkt till Business Management som beslutar vad som ska vara med i framtida produkter. Krav diskuteras direkt med respektive PU både när det gäller krav på GAS och speciella kundkrav. PU leverar sedan till Business Solution vilka ser hela systemet som en modell och en frisläppning även om det på grund av olika kundkrav i själva verket är många olika varianter. Varianter hanteras lokalt av PU med "marknadsparametrar".

Tidigare skedde marknadsutvecklingen nära marknaden. Det var bra på det sättet att de marknadsansvariga visste exakt vad som levererades till respektive kund. Nackdelen var dålig återanvändning av lösningar mellan olika kunder, vilket ledde till onödigt stora underhållskostnader efter frisläppning.

Nu är utvecklingen mer centralstyrdd. Enhetsansvariga för varje block av en viss typ hanterar de olika varianterna för sitt eget block vilket gör att man lättare kan återanvända gemensamma delar mellan de olika varianterna. Risken är nu i stället att man tappar marknadskontakten.

Felhantering

För felhantering utnyttjas ett verktyg kallat MHS (Modification Handling System). Testgrupperna (och även kunder) skriver felrapporter direkt i

MHS, vilka sedan halv-automatiskt skickas till rätt MHO (Modification Handling Office). MHO är en manuell växel som finns på varje LDC från vilken felrapporterna går ut till rätt enskild utvecklare.

Man kan hela tiden följa en felrapport genom systemet. Tex kan man få fram status på en rapport, hur många dagar varje person har haft den "på sitt bord", vem som just nu har rapporten, m m. Detta är väldigt bra och bl a därför används exakt samma hantering även för utvecklingsprojekt, då med andra bokstavs/nummer-serier. Att ha samma system gör också att alla känner till processen väl.

Det är viktigt att testledarna är tekniskt duktiga så att felrapporterna verkligen kommer till rätt LDC/person. Det finns dock alltid en risk för att det blir "långbänk" vid diskussionerna om vem som ska rätta ett fel, dvs mellan olika ansvarsområden. Ibland kan ett fel åtgärdas på många olika sätt, där de olika sättens ger olika mycket arbete för de olika ansvarsområdena. Enda sättet att lösa långbänken är genom att gå upp i besluts hierarkin och låta någon överordnad "peka" på den som ska göra felrättningen. Den distribuerade situationen, och då framförallt kulturella skillnader, visar sig här tydligt och kan innehåra att en annars onödig hög chefsnivå erfordras för att "peka".

Arkiv

Som register och arkiv utnyttjas koncernens PRIM (en stor innehållsförteckning som talar om vilka dokument som ska finnas i vilka versioner) och GASK. PRIM och GASK fungerar bra, se även beskrivningen i avsnitt 10.7, "Ericsson Microwave Systems AB". Business Solution ansvarar för att GAS arkiveras och registreras medan respektive PU ansvarar för sina egna produkter (som ju kan finnas i flera varianter vilket BS inte hanterar).

Källkod slutförvaras i ett källkodsbibliotek. Vid produktrelease (PRA i PRIM) görs en säkerhetskontroll om att alla dokument är lagrade rätt i GASK och i källkodsbiblioteken.

Förutom PRIM och GASK används DELTA. DELTA är ett stort projektbibliotek likt en jättepärm med uppslag där halvfärdiga dokument mellanlagras. Här kan alla (hela koncernen) även se status för alla dokument. Nya dokument och förändringar av dokument läggs in kontinuerligt vilket gör att övriga LDC kan följa utvecklingen. Mot slutet av projektet när dokumenten blir klara överförs de från DELTA till GASK. Före överföringen görs ett antal kontroller vilket är ett krav för att kunna lagra dokument i GASK. Till slut ska alla dokument finnas i GASK.

Beslutsorgan, dokument, m m

De olika beslutsorgan som finns är:

- Linjefunktion - linjechefer för respektive PU och BS. Är ytterst ansvarig för projektet. Kontroller sker genom styrgrupper inom varje PU och BS.
- Projektledning - projektledare från BS och från respektive PU. Kontrollerar projektet. Projektledarna träffas 1 ggr/månad för att stämma av status mot tidplan samt att koordinera konstruktion och test.

- Systemledningsfunktion - tekniskt kunniga på BS som för dialog med PU.
- CCB - en grupp inom projektet som godkänner avvikeler mot projektspecifikationen. Gruppens sammansättning beror på graden av avvikelse.

Erfarenheterna visar att videokonferens fungerar hyfsat om man känner alla, dvs man har träffat dem tidigare. Oftast fungerar dock telefonkonferens bättre och fysiska möten är att föredra. Om något allvarligt problem ska lösas ordnas nästan alltid ett riktigt möte.

Under projektet är det framförallt fyra dokument som är viktiga: IS, IP, FF och DPD. Under projektets inledande faser är i huvudsak IS (Implementation Sketch) och IP (Implementation Proposal) viktiga. Dessa dokument beskriver den tekniska lösningen på de krav som ställs mot objektet.

Utifrån IS och IP skrivs sedan en eller flera FF (Functional Framework) där informationen blir mer detaljerad så att konstruktionen kan starta. Utifrån FF kan sedan DPDer (Delivery Package Definition) göras upp.

En DPD definierar de nya programmen (filer) som ett nytt objekt består av. I DPDn kan man utläsa vilka andra program i andra PUar som skall vara med. Det är viktigt att allt i en DPD levereras vid samma tidpunkt och det är projektledaren som är ansvarig för att leverans sker i tid.

Generellt sett så fungerar hanteringen av ovanstående dokument bra. Alla kan tydligt se de olika ansvarsområdena. DPDn används för att få konsistenta frisläppningar med versioner av olika delprodukter som logiskt hör ihop. Dvs de hanterar aspekter som är viktiga vid en distribuerad situation.

När det gäller sekretess förlitar man sig på IBMs stordatormiljö med applikationer som t ex PRIM, GASK och DELTA. Inget överförs öppet över Internet och Intranet används endast för harmlös information

10.10 Factum Elektronik AB

Intervju, 1998-10-22

Thomas Nilsson, CM ansvarig och projektmentor för DBS100-projektet

Projekt och produkt

DBS100 är ett relativt litet men mycket distribuerat projekt bestående av 7 personer: två i Stockholm, en i Danmark, en i Norge, två på Factum i Linköping och Thomas på Rational Software i Linköping. Det startade i våras (-98) och den första leveransen är i början av november, dvs efter ca 4-5 manår. Efter frisläppningen tar Factum över produkten och kommer hantera underhållet lokalt i Linköping.

Produkten är ett system för att distribuera data och information över vanlig radio, dvs genom broadcast. Implementationen följer den internationella standarden DAB (Digital Audio Broadcast) som kan ses som

nästa generation av RDS-systemet som finns i de flesta radioapparater idag.

Tänkta tillämpningar är dels multimediatillägg till vanliga radiotransmisioner, t ex väderkartor eller bilder att komplettera reklaminslag, dels organisationer som vill nå ut till sina medlemmar, där ett exempel kan vara utsändning av nya prislistor till ICA-handlare. En annan möjlighet är att sända datatrafik över (från) Internet. I det fallet måste begäran om data ske på annat sätt, t ex med vanligt modem över telenätet.

Produkten omfattar protokollsdelens och är tänkt att finnas på små radiostationer eller som en del av en större station som t ex Sveriges Radio.

Utvecklingen

Då projektet är litet men distribuerat har man inriktat sig på en informell och effektiv organisation. En arkitektgrupp om tre personer gjorde själva systemkonstruktionen och beslutar i strategiska frågor under projektets gång. CM-ansvarig beslutar om när taggningar (interna frisläppningar) ska ske och en projektledare ser till att tidsplanen följs samt fördelar resurser. Beskrivning i form av CM-plan finns ej.

Som CM-verktyg används CVS med ett centralt arkiv på en server för alla typer av dokument. Arbetet på de olika siterna sker genom att man tar en arbetskopia av hela CVS-trädet enligt metoden i CVS, varefter man arbetar lokalt. Vid commit checkas alla ändringar in och uppdaterar arkivet. Inga projektgrenar, förutom för test, används utan alla arbetar direkt på huvudgrenen, vilket gör att alla ändringar syns på huvudgrenen direkt. Fördelen är att man tidigt testar ny funktionalitet, medan nackdelen är att buggar incheckade på huvudgrenen riskerar att förstöra testmiljön för de övriga. Detta problem är dock inte så stort på grund av CVS strategi där varje utvecklare väljer när man vill ha uppdateringar i sitt arbetsbibliotek.

För att minska nackdelarna testas alltid en modul före uppdatering, en överenskommen process, utan något formellt stöd i form av CM-plan, som fungerar bra. Praktiskt hanteras det genom att varje modul har ett testbibliotek med ett huvudprogram för modulen,testdata och drivrutiner (testscenarier). Huvudprogrammet är generiskt så det är enkelt att utöka testerna genom att endast lägga till nya scenarier och testdata.

För att minska riskerna att samma filer ändras samtidigt på de olika siterna (arbetet sker ju isolerat på varje site) har varje modul en ansvarig. Vill man ändra i någon annans modul ber man antingen den ansvariga göra ändringen eller så får man godkännande att själv göra ändringen. Tack vare att projektet är litet och att all kommunikation kan ske snabbt och informellt via telefon och mail uppfattas inte metoden som för tungord. Detta är i sin tur möjligt då inga kulturella skillnader finns och alla är i samma tidszon.

Hela produkten byggs varje vecka. Man tar fram det man vill bygga, sätter en "tag" på hela systemet (alla filer) och skapar en bygg-gren. Eventuella små ändringar som måste göras för att få ett komplett bygge görs i grenen. Utvecklingen kan under tiden fortsätta i huvudgrenen. I de fall fel upptäcks som även bör komma med i huvudgrenen hanteras det

manuellt. Varken formell ändringshantering eller ambitiös merge används utan ändringarna dupliceras genom "copy-paste".

Medvetenheden om vad andra arbetar med är ganska stor då det är ett litet projekt. För att ändå ge så mycket stöd som möjligt för awareness utnyttjas de möjligheter som finns i CVS, vilket ger två nivåer av awareness:

- Den vanliga funktionaliteten i CVS genom att man ser vad som hänt när någon gör update. Ingen ytterligare metadata används.
- För speciellt viktiga filer utnyttjas s k "watchers". Det innebär att vanlig checkout resulterar i en enbart läsbar kopia av versionen. Vill man få skrivrättigheter måste man även göra en "edit". Detta innebär att man meddelar att man tänker modifiera filen. Andra utvecklare som är intresserade av just den filen har möjlighet att registrerar sig och prenumerera på den här typen av meddelanden, vilka kan skickas både vid "edit", "unedit" och "commit". "Viktiga filer" är typiskt gränssnittsbeskrivningar men även binärer, då de är svåra att merga och därför inte bör ändras parallellt.

Då projektet är ganska litet och alla utvecklare verkligen arbetar i samma projekt och inte i olika delprojekt har man inte tyckt det vara nödvändigt att införa någon formell ändringshantering. Dessutom har ingen frisläppning gjorts än så det är endast kravändringar och fel som upptäcks under test som behöver behandlas (ingen felrapportering från användare). Då ett fel upptäcks tar man kontakt direkt med den utvecklare man tror är ansvarig.

Inget speciellt arkiv används utan man använder taggar i CVS.

Sekretessen består endast av att skydda produkten. Mellan siterna sänds all data okrypterad över Internet.

En integrerad utvecklingsmiljö med Visual C++ på Win95-plattform används. Konsistenta utvecklingsmiljöer erhålls genom att inblandade produkter används i samma versioner på alla siter.

Projektet började från scratch med att använda CVS så någon problematik vid införande har det aldrig varit. De inblandade parterna är nöjda med lösningen och erfarenheter från projektet kommer troligen att tas in i andra projekt.

10.11 Kockums Computer Systems

Intervju, Malmö 1998-03-31

Krister Erlansson

Företaget

Kockums Computer Systems (KCS) grundades 1977 och har idag ca 150 anställda varav ca 60 är utvecklare. Huvudkontoret är i Malmö med dotterbolag i Tyskland, Japan, Sydkorea, England och USA. Produkten, TRIBON, ett produktionssystem för fartygstillverkning, används av mer än 270 varv i 38 länder, bl a Asien, Australien, Europa och Nord- och

Sydamerika. TRIBON är ett CAD/CAM/CIM-system bestående av ett 50-tal olika produkter integrerade till ett komplett system.

Utvecklingen är avancerad med avseende på CM då produkten är portad till många plattformar (VAX, ALPHA, HP, IBM, Sun, och PC), på vilka många operativsystem samtidigt underhålls (t ex HP-UX9, HP-UX10, AIX3, AIX4). Dessutom används många språk (PL/I, Fortran, C, C++, UIL och assembler), vilka givetvis också existerar i många versioner. Detta sammantaget tillsammans med det faktum att utvecklingen är distribuerad på tre siter (Malmö, Tyskland och England) gör att KCS har utvecklat en väl fungerande CM-process. I Malmö har de ca 10 VOB:ar och 30000 filer.

Fram till 1994 användes ett egentillverkat CM-verktyg. Verktyget, som kallas MMS, klarade i stort sett de krav man då hade men fanns endast på VMS. Då allt fler kunder gick över till Unix letade man efter ett kommersiellt alternativ och bestämde sig för ClearCase, vilket man gradvis gick över till med start 1994.

Förutom att MMS endast fanns på VMS gav det inget stöd för distribuerad utveckling. Utvecklare från dotterbolagen loggade in på datorer i Malmö och körde över en höghastighetslinja som ständigt var uppkopplad. Det var en mycket dyr lösning, som dessutom knappast gick att använda för att testköra de grafiska programmen pga för dålig prestanda.

Idag används ClearCase och ClearCase MultiSite tillsammans med egenutvecklade script. Intranet och web-teknologi används flitigt både för att komma åt information genererad av ClearCase men även för dokumentation av CM-process och hur-man-gör listor.

Utvecklingsprocessen

En central kodbas för hela TRIBON innehåller all funktionalitet för alla plattformar. Projekt utvecklas i speciella projektgrenar vilka merges in till huvudgrenen efter tester och kvalitetskontroll. Inkrementell utveckling har länge varit KCS:s arbetsätt vilket fås genom generiska regler vilket ger partiellt bundna konfigurationer (se avsnitt 3.3). Exempelvis använder normalt ett projekt den senaste incheckade versionen på huvudgrenen, vilket medför att allt incheckat på huvudgrenen direkt testas av dessa projekt. I sällsynta fall förekommer det även att projekt använder sig av versioner på andra projektgrenar, vilket gör att dessa delar testas mot varandra ännu tidigare än då projektet väntar tills de finns på huvudgrenen.

Utveckling sker på tre siter, Malmö, England och Tyskland. Huvudutvecklingen sker i Malmö och utvecklingsgruppen i England ansvarar för delprodukten Initial Design och Tyskland för delprodukten Work Preparation.

Varje utvecklingskontor äger sina grenar. För de produkter som respektive kontor ansvarar för, äger de också huvudgrenen. Dvs de ansvarar själva för integrationen av sin produkt med övriga produkter.

Sekretess

Inga speciella åtgärder när det gäller sekretess. Alla utvecklare på alla siter kan komma åt all källkod. De begränsningar som finns är enbart de

som introducerats av ClearCase MultiSite, dvs att man endast kan skapa nya versioner på en gren ägd av den site (VOB) man är på.

All kommunikation mellan sittorna, t ex synkroniseringsdata, sker idag okrypterat på "egna" ledningar där möjligheten för "sniffning" är mycket liten. KCS kommer dock snart övergå till att använda internet, men då krypterat.

En tredjepartsleverantör utvecklar och säljer stöd för kundanpassning av TRIBON. Deras produkt är väl integrerad med TRIBON men utvecklingen sker helt avskilt från varandra. KCS tillhandahåller gränssnitt mot vilka de hela tiden är bakåtkompatibla vid nya releaser. Några sekretessproblem finns dock inte då ingen källkod delas.

Det är dock vanligare att kunderna själva gör sin anpassning istället för att anlita en tredjepartsleverantör.

Ändringshantering

För felrapporter används ett externt system, HelpLine, som registrerar alla rapporter i en gemensam databas. Rapportering av fel tas om hand av Local Support i det land som kunden kommer ifrån, där felet först reproduceras. Vissa fel kan Local Support själva klara ut med kunden (användarfel, fel i kundens miljö, etc.). Resterande fel rapporteras vidare till respektive produktansvarig, som gör en grundligare bedömning och ger ett svar som skickas tillbaka till Local Support, som i sin tur svarar kunden. Det är alltså den produktansvarige som har överblicken, men Local Support har full insyn i Help-Line-databasen och genom den kan de få en viss överblick.

KCS erfarenhet visar att detta ger dem de snabbaste svarstiderna, genom att Local Support känner ansvar för att kunden blir hjälpt snabbt och att inga fel blir bortglömda.

Inte endbart fel utan även små tillägg hanteras med HelpLine. Större ändringar och kundprojekt utförs vanligtvis utan att varje enskild ändring registreras i HelpLine. Då används specifikationer i någon form.

Varianthantering

Varianter pga av olika funktionalitet utvecklas endast i undantagsfall och då i speciella projektgrenar. Endast huvudgrenen har garanterad överlevnad. Förr eller senare tas projekt- och flerrätningsgrenar bort vilket innebär att funktionaliteten då, för att överleva, måste mergas in till huvudgrenen.

Varianter pga portning till olika plattformar hanteras helt med villkorlig kod, dvs plattformsberoende och plattformsberoende kod blandas i samma filer, vilket underlättar att hålla funktionaliteten på de olika plattformarna konsistent.

Parallell utveckling/Awareness

Normalt har projekt sina egna utvecklingsgrenar medan utvecklare inom samma projekt arbetar på samma gren. På så sätt kan projekten arbeta helt parallellt. Utvecklare i samma projekt kan också arbeta parallellt, dock ej i samma fil samtidigt.

KCSs inkrementella utvecklingsstrategi med tidig testning förhindrar projekt att divergera för mycket, även då de utvecklas parallellt. För att

minskar nackdelarna med denna strategi görs integration av tillägg som kan misstänkas ge problem först omvänt, dvs merge från huvudgrenen till projektgrenen och test där. Sedan, om allt fungerar, kan mergen till huvudgrenen göras, vilken ofta då blir trivial. Test mot övriga projekt har dock ej gjorts, utan görs nästa gång respektive projekt själva bygger och testar.

Medvetenheten om vad som händer i andra projekt innan de merges till huvudgrenen är oftast låg.

KCS har gjort egen implementation utöver ClearCase för att underlätta åtkomsten av awareness-information. T ex kan man från ClearCase generera information om vilka grentyper som finns och vilka olika konfigurationsbeskrivningar som används. Genereringen tar dock rätt lång tid vilket gör att dessa funktioner inte används i den utsträckning de borde. Periodisk generering och utläggning av resultatet på intranet där snabb åtkomst är möjlig löser en del av problemet.

Datalagring

ClearCase används även som arkiv. Ordinär backup på alla siter, även för de VOB:ar som är replikerade.

Releasehantering och distribution

Speciella make-script genererar systemet för en given plattform. Header-kommentarer ser till så rätt kompilator startas och att rätt villkorlig kod kompileras. All funktionalitet inkluderas i alla releaser. Kundspecifika konfigurationer där t ex endast en delmängd av all funktionalitet "levereras", hanteras med konfigurationsfiler och licensserver. På så sätt minskar man antalet byggda system och behöver endast hantera varianter på konfigurationsfilerna.

Tips och fallgropar

Övergången från det egna CM-verktyget till ClearCase gjordes stegvis. Under ca 6 månader var ClearCase endast ett parallellt backup-system. Övergången gjordes sedan plattform för plattform. Dessutom bevarades den gamla arkitekturen och man ansträngde sig för att, med script, implementera de populära stödfunktionerna även i ClearCase, vilket säker bidrog till att hela övergången gick smidigt.

Då ClearCase inte finns på VMS, fick de lite praktiska problem med montering av diskar mellan Unix och VMS. Även med fungerande montering gjorde avsnaknaden på läsningsmekanism att ändringar under kompilerings- och länkningssessionerna (VMS resp. Unix/ClearCase) gjordes samtidigt. Skedde så, kunde filer försvinna. CM-systemet bör alltså finnas på alla de plattformar produkten stöder.

Framtiden

KCS går över allt mer till att endast utveckla på NT. Speciella portningsgrupper hanterar portningen till de övriga plattformarna, vilka blir allt färre. En NT version kommer dock först om ett till två år.

Vid vissa tillfällen besöker utvecklare någon av de andra utvecklingsställen för att t ex hjälpa till vid testning och integrering. Förutom detta arbete vill de även kunna arbeta vidare med sitt "hemma"-arbete.

Pga av restriktioner i ClearCase MultiSite kan de nu inte arbeta på sin vanliga utvecklingsgren utan ett specialarrangemang måste göras. Detta är givetvis möjligt att göra, men ett starkare stöd för denna typ av arbetsprocess välkomnas i framtiden.

10.12 Saab Gripen

Intervju, Linköping 1998-09-07

Mikael Djurberg, projektledare för IT, sekretess

Torbjörn Fransson, distribuerad utveckling och CM

Björn Svensson, utvärdering och införande av CM-verktyg

Distribuerad systemutveckling - ej distribuerad programutveckling

Saab Gripen och British Aerospace (BAe) samarbetar vid utvecklingen av Gripen. De har en gemensam organisation med gemensam chef över både svensk och brittisk personal. Utveckling sker både i Linköping och i Brough och en skyttel går varje vecka mellan städerna för att underlätta samarbete. Tidigare skedde all utveckling i Linköping men delar ska efterhand flyttas till Brough.

Distributionen mellan Linköping och Brough är enligt fallet Sammanhållna grupper på olika platser. Det finns även fallet Outsourcing i form av underleverantörer/partners som utvecklar, har egna testmiljöer för egna specifikationer och levererar binärer. Källkod och dokumentation skickas med men kompileras och länkas ej. Hemarbete sker inte on-line utan endast off-line och då aldrig med hemlig information.

Ett aktuellt problem är att de inte kan arbeta på hemlig information i Brough. Det är ett juridiskt problem då det blir "export av krigsmateriel" om någon försvarsheilig information lämnar landet. I det fall det måste ske, görs det med diplomatpost vilket är mycket för långsamt att använda för normal utveckling. I dagsläget kan man därför säga att de har distribuerad systemutveckling men inte distribuerad programutveckling.

Moduler är den minsta separatkompilerade enheten i systemet. Det finns alltid en ansvarig för varje modul och det är endast den personen som får skapa nya versioner av modulen. Det finns även konfigurationsansvariga vilka är de enda som kan skapa nya versioner av sina konfigurationer bestående av nya versioner av de ingående modulerna.

Hantering av ändringar

För att få göra en ändring måste det finnas en ändringsbegäran. I det CM-verktyg som används idag, JACC, finns dessutom stöd för att hålla ihop sammanhängande ändringar. En ny version av en modul kan inte ingå i någon konfiguration förrän alla andra moduler modifierade i samma ändringsbegäran också är klara. På så sätt undviker man inkonsistenta konfigurationer där bara delar av ändringar är genomförda.

Uppdatering av gränssnittsspecifikationer följer en lite annorlunda process. Då flera utvecklare får ändra i samma specifikation är en hårdare synkronisering än som används med vanliga moduler nödvändig. För att få göra en ändring i ett gränssnitt måste man fysiskt "hålla" pärmén

med dokumentation, dvs en helt manuell metod där pärmen fungerar som ett slags stafettpinne används.

Sekretess och säkerhet

Saab är nu i ett skede då nya metoder, processer och verktyg tas fram för att stödja distribuerad utveckling inklusive de sekretessproblem som finns vid hanteringen av försvars hemlig information. Det egenutvecklade systemet JACC ska bytas ut och ersättas av PCMS där roller och åtkomsträttigheter bildar en bas.

Krav- och hotbilden för säkerheten är sammanställd utifrån: egna krav (företags hemligheter), svensk lag och kundkrav (t ex FMV). Exempel på krav från en köpare är att hålla viss systemfunktionalitet hemlig, t ex motmedel, vapen och sensor teknik. Säkerheten omfattar även dataintegritet, dvs att behålla informationen konsistent och ej modifierad eller förvanskad.

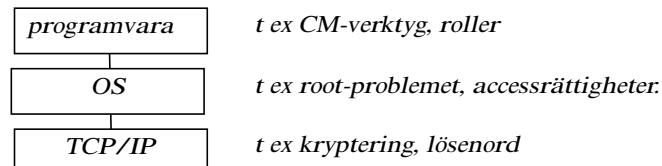
Sekretesskraven är ofta, framförallt i en distribuerad miljö, motstridiga med andra krav som återanvändning av kod och resurser. Öppna nätverk som underlättar kommunikation och synkronisering av siter ställs mot speciella rättigheter på information. En annan viktig del i implementerandet av sekretess är att ha informationsägare på hemlig information. Han/hon ansvarar för informationens hela livscykel, att den är rätt klassad, ges till rätt person och att det endast är godkända system som hanterar informationen. Ett ansvar som måste klaras även då informationen finns eller flyttas mellan flera siter.

Tre säkerhetsfunktioner kan definieras vilka, om de hanteras rätt, leder till en bra faktisk säkerhet.

- logiska funktioner: hård- och mjukvara, t ex logiska nätverk, TCP/IP, identifiering och autentisering, åtkomstkontroll, spårbarhet, kryptering, m m.
- fysiska funktioner: t ex skalskydd, larm, brandskydd, elmiljö, m m.
- organisorisk funktioner: roller och ansvar, personkontroll, utbildning, regler och rutiner, m m.

Ett annat sätt att dela upp säkerhetsimplementationen är enligt de tre nivåer som beskrivs i figur 37. För en bra säkerhet måste alla nivåer vara säkra.

Programvara ska vara känd och kontrollerad innan den används i systemet. Vid val av t ex mjukvara för CM bör den lämpa sig för flera olika typer av miljöer. Den måste även fungera via brandväggar och liknande



Figur 37 Säkerhet på tre nivåer

filter. Vidare ska den hantera åtkomsträttigheter bra mot operativsystemen. Man ska t ex inte behöva flera konton om man använder NT och Unix samtidigt.

Operativsystem (OS) ska vara "evaluerade" av någon säkerhetsorganisation, t ex ITSEC standard. Det kan vara lämpligt att införa någon form av säkerhetsprogramvara som tillägg till OS. T ex är Unix grundskydd idag för lågt och medför en klumpig och grov hantering av åtkomstskydd. NT är svårt att administrera.

TCP/IP Kryptering på TCP/IP-nivå kan förhindra felaktig åtkomst av data, lösenord, IP-adresser m m.

För att uppnå bästa möjliga säkerhet är det mycket viktigt att få en samklang mellan de tre nivåerna; programvara, OS och TCP/IP.

Programvara och OS ska säkerställa att det går att implementera roll- och ansvarstilldelning. Viss programvara hanterar själv behörighetskontroll som inte är kopplad mot OS. Detta medför stora problem vid en eventuell flytt av data till en annan site. Det medför även stora problem under underhåll vid distribuerad utveckling.

Det finns även verktyg som kan hantera att delar av ett dokument är hemligt. Vill man kunna låta resten av dokumentet vara öppet tillgängligt, måste säkerheten lösas på applikationsnivån ("programvara" i figuren). Operativsystemet hanterar hela dokument (filer) och måste betrakta allt enligt den hemligaste delen av ett dokument.

TCP/IP-hantering (krypto etc.) är viktigt men inte hela lösningen på problemet. Detta har idag ett mycket stort fokus. En fungerande brandvägg gör dock inte att det blir bättre ordning. Roller och ansvar förbättras inte.

Saab Gripen arbetar mycket med den översta nivån för att få tydliga roller med verktygsstöd. Målet är roller med tydliga rättigheter. Ett verktygsstöd som stöder dessa roller även i en distribuerad miljö är målet.

Arkiv

JACC används även för slutförvaring av både programvara och dokument. Definitionen av ett arkiv är att det har en kontrollerad fysisk lagring och att levererad produktdata hanteras av en produktbibliotekarie. Det är alltså viktigt att ansvaret för arkivet ligger hos en produktbibliotekarie och inte kvar hos projektledaren efter frisläppandet. Däremot kan själva dokumenten ligga kvar i samma verktyg bara det är ordentligt "taggat" och dokumenterat.

Införande

Saab använder sig idag av många olika verktyg för de olika systemdelarna, bl a Continuus, ClearCase, JACC, RCS, CVS och CMS. EMPIRE-projektet startades 1996 för att ensa utvecklingsmiljön för Gripen och införa gemensamma metoder, processer, verktyg, utbildningar, m m.

Testfall med både kravspecifikation och användningsfall togs fram. T ex hade avdelningen för Styrsystem tre grundkrav: (1) säkerhet, (2) parallell utveckling, (3) ändringshantering (med hierarkisk livscykel). Saab utvärderade olika verktyg utifrån testspecifikationen. PCMS valdes.

Ett skarpt pilotprojekt, GECU (Generic Electric Control Unit), med ca 15-20 personer plus underleverantörer ska bli första testen. Projektet är nytt och utan historia.

10.13 Telelogic

Intervju

Piotr Nestorow, Malmö 1998-05-18 och 1998-11-17

David Prather och Magnus Håkansson, Malmö 1998-11-17

Företag och produkt

Telelogic utvecklar och säljer utvecklingsverktyg för design, implementering och testning i SDL. Produkterna heter SDT och Itex.

Utvecklingskontor finns i Malmö, Uppsala och Tyskland med ca 25, 5 respektive 16 utvecklare. De har även partners, dvs de arbetar på uppdrag av Telelogic, i Tyskland. Huvudutvecklingen sker i Malmö där större delen av SDT utvecklas. I Tyskland utvecklas en kodgenerator (C-micro) för små målsystem med lite minne. I Uppsala utvecklas delprodukten Itex, vilket är en miljö för TTCV i vilken man skriver testsviter. Itex är väl integrerat med SDT både vad gäller användning och utveckling.

Dessutom finns säljkontor även i USA, Frankrike och England.

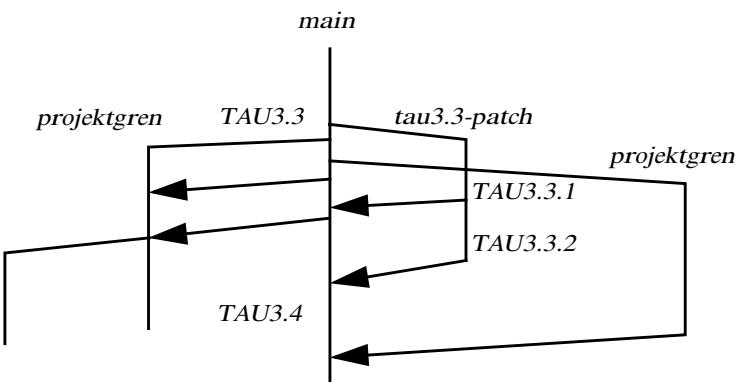
Verktyg och utvecklingsprocess

TeleLogic använder sedan -97 ClearCase som CM-verktyg. Året efter började de även använda ClearCase MultiSite för synkronisering av Malmö och Uppsala, något som fortfarande utvärderas. All kommunikation mellan Malmö och Uppsala sker på hyrda linjer.

Synkroniseringen mellan Malmö och partners sker manuellt med ftp ca en gång per vecka. Kommunikationen sker okrypterat över Internet. För varje partner finns en kontaktperson i Malmö som ansvarar för att ändringarna förs in i ClearCase på rätt sätt (i deras egen gren). I övrigt skiljer sig inte denna utveckling från den inom huset, utan samma CM-process kan användas.

Figur 38 visar ett exempel på en fils olika grenetyper (ex. tau3.3-patch) och satta "taggar" (ex TAU3.3). Den mesta utvecklingen sker direkt på huvudgrenen (main). För underhåll tas grenar ut för varje frisläppt version (ex. tau3.3-patch). Buggrättnings görs i dessa grenar och, om relevant, även i main (antingen manuellt eller genom merge enligt figuren). Från underhållsgrenen levereras också patchar till kund. För långsiktiga projekt som t ex att ge stöd för asiatiska språk, används också grenar i vilka dessa projekt kan arbeta mer ostört, t o m över flera frisläppningar. Dock har företaget som regel att även dessa grenar med jämna mellanrum ska synkronisera sig mot main för att inte divergera för mycket.

Generellt används olika grenar beroende på vilken frisläppning ändringarna planeras ingå i. Om ett projekt ska bli klart till nästa frisläppning görs ofta utvecklingen direkt i huvudgrenen (utvecklaren kan själv bestämma). För längre projekt som pågår över frisläppningstidpunkter skapas en gren i vilken arbetet kan pågå utan att påverka fris-



Figur 38 Visar ett exempel på hur olika grenar används både för fejrättning och för ny funktionalitet.

läppningen. När projektet är klart görs merge mot huvudgren och grenen avslutas.

Produkten är uppdelad i moduler där utvecklare mestadels arbetar i ”sin egen” modul. Moduluppdelening är extra tydlig för det som utvecklas på uppdrag av partners. Dessa arbetar helt i sin egen modul och använder oftast en tidigare frisläppt version av verktyget som testmiljö. Sluttester sker i Malmö eller Uppsala beroende på produkt. Partners får använda sina tidigare CM-verktyg och behöver alltså inte anpassa sig till TeleLogic.

Alla ändringskrav behandlas av ett CCB bestående av personer från Malmö, Uppsala och säljkontoren. Efter behandling prioriteras de och läggas i en databas.

Årsmodeller med frisläppning två gånger per år tillämpas. Före införandet av årsmodeller utvecklade man tills den nya funktionaliteten var färdig, då frisläppningen gjordes. Framförallt krav från kunderna ledde till modellen med årsmodeller, vilket underlättar mycket för marknads- och supportavdelningarna. För utvecklarna är det däremot både på gott och ont. En fördel är att alla känner till de fasta tidpunkterna och kan planera därefter. Nackdelen är att man på ett tidigt stadium måste bestämma om en viss ändring ska ingå i nästa frisläppning, eller i en senare. Ett felaktigt beslut då resulterar i merarbete för att antingen sent försöka integrera in funktionalitet eller för att ta bort ännu ej buggfri funktionalitet från huvudgrenen.

Heterogen utvecklingsmiljö

Utvecklingen i Malmö och Uppsala sker i en heterogen miljö med parallell utveckling på Solaris och NT. På så sätt kan man lättare göra frisläppning på båda plattformarna samtidigt. Portning sker även till andra plattformar, t ex Win95 och en rad Unixdialekter.

Utveckling på två plattformar innebär också att ClearCase ska fungera både på NT och på Unix, vilket innebär en del problem. Bara att få identiteter och grupper att fungera tillsammans i båda miljöerna är svårt. Det gör det också besvärligare för utvecklarna att följa den fastställda utvecklingsprocessen. Varje utvecklare ska innan de gör checkin

(vilket ju oftast görs direkt på huvudgrenen) testa sina ändringar för att inte orsaka onödiga fel för de övriga. Detta måste göras för bågge plattformarna. Om arbetet utförts i en vy på NT är det svårt att komma åt dessa filer från Unix, och det går således inte att bygga och testa under Unix (tvärtom går med hjälp av Samba). Resultatet är att man först gör incheckningen och möjligen därefter testar även för Unix.

Arkiv

Flera ClearCase VOB:ar används: leverans- (gemensam), källkod- (sitespecifik), verktygs- (sitespecifik), validering- och kursmateriel-VOB.

Genom att ha en verktygs-VOB i vilken alla utvecklingsmiljöer (inklusive kompilatorer) hanteras klarar man att hålla alla utvecklingsmiljöer konsistenta. Vid frisläppningen kopierar man dessutom aktuell miljö till en hårddisk som arkiveras.

Felhantering

Nyligen har en kommersiell produkt, Heat, köpts in för att hantera felrapporter från kund och i framtiden även från testare. Alla utvecklare ska använda html-baserade ingångar för att komma åt aktuella rapporter. Supportavdelningen som ensamma har utvärderat verktyget är nöjda. Utvecklarna ska nu se hur de ska koppla ihop Heat och ClearCase för att få spårbarhet mellan felrapporter och åtgärder i dokument lagrade i ClearCase. Tidigare användes ett av utvecklarna framtaget system med en databas vilken projektledarna hanterade. Det fungerade bra för utvecklingsavdelningen men endast lokalt och det gav inget direkt stöd för supportavdelningen.

Referenser

- [Aalborg] <http://www.cs.auc.dk/research/PS/>
- [Aarhus] <http://www.daimi.aau.dk/~hbc/Ragnarok.html>
- [AM97] U. Asklund och B. Magnusson. A Case-Study of Configuration Management with ClearCase in an Industrial Environment. I *Proceedings från SCM7 - International Workshop on Software Configuration Management*, R. Conradi (Ed.), Boston, maj 1997, LNCS, Springer Verlag.
- [App98] Links to Software Configuration Management on the World Wide Web. <http://www.enteract.com/~bradapp/links/scm-links.html>
Brad Appelton. 1999.
- [BGD96] C. Burrows, G. W. George, and S. Dart. Ovum Evaluates: Configuration Management, Ovum Limited, London 1996
- [BW98] Clive Burrows and Ian Wesley. Ovum evaluates: Configuration Management, Ovum Ltd 1998. ISBN 1-898972-24-9
- [CSCW] Computer Supported Cooperative Work - An International Journal. Kluwer Academic Publishers.
- [CSCWyp] The UnOfficial Yellow Pages of CSCW. <http://www.telekooperation.de/cscw/yp/>
- [Ced93] Per Cederqvist. Version Management with CVS. Available from infosignum.se, 1993.
- [Cla95] Dave St. Clair: Continuus/CM vs. ClearCase, URL: <http://sunsite.icm.edu.pl/sunworldonline/swol-07-1995/swol-07-cm.html>, SunWorldOnline, 1995.
- [Clear98] <http://www.rational.com/products/clearcase>
- [Con98] <http://www.continuus.com>
- [Conr98] <http://www.idt.unit.no/~reidar>
- [CW97] R. Conradi och B. Westfechtel. Towards a Uniform Version Model for Software Configuration Management. I *Proceedings från SCM-7 [SCM-7]*. 1997
- [Cyc99] Web-sidor bl a om CVS. <http://www.cyclic.com>
- [Dar90] S. Dart. *Spectrum of Functionality in Configuration Management systems*. Technical report CMU/SEI-90-TR-11, Software Engineering Institute, Carnegie Mellon Institute, december 1990.
- [Dar97] S. Dart. To Change or Not to Change. *Application Development Trends*, juni 1997. T ex från <http://www.truesoft.com>.
- [Deg98] Lars Degerstedt. Klarar era produkter Internet? ...när kunden vill styra, programmera och uppdatera dem via Internet. Sveriges Verkstadsindustrier. 1998
- [ECSCW97] Proceeding of the Fifth European Conference on Computer Supported Cooperative Work. Kluwer Academic Publishers. ISBN 0-7923-4638-6.
- [ELS98] Elsinore Technologies, Inc., <http://www.elsitech.com>
- [EPOS] <http://www.idt.unit.no/~epos/>
- [Exco99] http://www.excosoft.se/conf/conf_overview.html
- [Fei91] P. Feiler. *Configuration Management Models in Commercial Environments*. Technical report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie Mellon Institute, mars 1991.
- [FSE6] Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering (FSE-6), Florida, USA, november 1998. ACM Software Engineering Notes 6(23).

- [GMD] CSCW Group, Institute for Applied Information Technology (GMD FIT). Sankt Augustin, Tyskland. <http://bscw.gmd.de/>
- [HVT96] J. J. Hunt, K-P Vo och W. Tichy. An Empirical Study of Delta Algorithms. I Proceedings från SCM-6 [SCM-6], 1996.
- [Hoe99] André van der Hoek. Configuration Management Yellow Pages: http://www.cs.colorado.edu/users/andre/configuration_management.html
- [ICSE97] Proceedings of the 19th International Conference on Software Engineering. Boston, Massachusetts, USA. 1997. ISBN 0-89791-914-9.
- [JS98] JavaSafe 1.0 User's Guide. JavaSoft, Inc. <http://java.sun.com/products/javasafe/index.htm>
- [Lund] Lunds Tekniska Högskola. Institutionen för datavetenskap. Programvaruteknik. <http://www.cs.lth.se/Research/ProgEnv>
- [MM93] S. Minör och B. Magnusson. A Model for Semi-(a)Synchronous Collaborative Editing. I proceedings från Third European Conference on Computer Supported Cooperative Work, Milano, Italien, 1993. Kluwer Academic Publishers.
- [Mol98] Pascal Mollis webbsidor om CVS. <http://www.loria.fr/~molli/cvs-index.html>
- [MSC98] Microsoft Corporation, <http://msdn.microsoft.com/ssafe/>
- [MSF98] MainSoft Corporation, <http://www.mainsoft.com>
- [PCMS99] www.sql.com
- [POW98] TechExcel, Inc., <http://www.powertrack.com/>
- [RBI95] W. Rigg, C. Burrows and P. Ingram: Ovum Evaluates: Configuration Management Tools, Ovum Limited, London, 1995
- [Roe75] Roekind, M. J., The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364-370, December 1975.
- [SCM-6] Proceedings från Software Configuration Management. Ian Sommervielle (Ed.) ICSE'96 SCM-6 workshop, Berlin, Tyskland, mars 1996. Springer LNCS 1167.
- [SCM-7] Proceedings från Software Configuration Management. Reidar Conradi (Ed.), ICSE'97 SCM-7 workshop, Boston, MA, USA, maj 1997. Springer LNCS 1235
- [SCM-8] Proceedings från System Configuration Management. Boris Magnusson (Ed.), ECOOP'98 SCM-8 symposium, Bryssel, Belgien, juli 1998. Springer LNCS 1439
- [SCM-9] Symposium on System Configuration Manatement. Toulouse , Frankrike den 5-7 september 1999. <http://www.cs.colorado.edu/~andre/scm9/>
- [SISU97] Swedish Institute for Systems Development (SISU). <http://vw.sisu.se/>
- [SMS] *Teamware Users' Guide*, SunPro Manual Set. Sun Micro Systems, Mountain View.
- [SOF98] SourceOffSite, Inc., <http://www.sourceOffSite.com>
- [Sun99] <http://www.sun.com>
- [SWT95] "Remote Software Development Using CodeManager". [<http://www.sun.com/workshop/teamware/wp-teamware/sep95/remote.html>]
- [SWT99] SUNs webbsidor om Sun Workshop Teamware. <http://www.sun.com/workshop/teamware>

176 Appendix - Tools and Interviews (in Swedish)

- [Wib97] D. Wiborg-Weber. Change Sets Versus Change Packages: Comparing Implementations of Change-BasedSCM. I proceedings från SCM-7 International Conference on Software Configuration Management, Boston, MA, USA, maj 1997. LNCS. Springer.