The Impact of Global Software Development on Software Configuration Management

Kaisa Uotila

University of Tampere
Department of Information Sciences
Master of Science Thesis
May 2003

University of Tampere
Department of Information Sciences
Kaisa Uotila: The Impact of Global Software Development on Software Configuration Management
Master of Science Thesis, 69 pages
May 2003

Abstract
In today's large and complex software projects, companies have understood the importance of software configuration management. The ability to manage changes effectively is a major key to successful projects. At the same time, the global software development has become common. Virtual corporations, usage of subcontractors, and starting a new development in a country, where the labour costs are cheaper, are few examples of the reason why software corporations are going global. The aim of this study is to analyse what kinds of impacts the emerging trend of global software development has on software configuration management systems.

The main conclusion is that the role of software configuration management is greater in global software projects than in local projects. Global software projects have extra requirements on software configuration management that can be divided in three groups: security, reliability, and ease of use. In an ideal situation, the used software configuration management tool supports all these requirements. However, the software company needs also to define processes to the software configuration management system to overcome all of the requirements.

Keywords: software configuration management, SCM, change control, global software development, distributed development, geographical distribution, parallel development, concurrent development

Acknowledgements

I would like to take this opportunity to thank my employer, Nokia Networks, for providing me the resources to write this thesis. All my former colleagues at the software configuration management teams deserve my gratitude for the challenging and inspiring working atmosphere. A special thanks goes to Katri Saarinen, who has endlessly supported and instructed me with this thesis. A word of thanks goes also to my present colleague and friend Johanna Välimäki for commenting my text.

At the University of Tampere, I wish to thank Seppo Visala for helping me to get started and for the encouragement at the early stages of this project. I'm also grateful to professor Jyrki Nummenmaa, who acted as my supervisor and gave me valuable advice and comments at the final stages of this work.

Finally, warm thanks to my family. Thank you to my parents, who have always supported me with my studies, and who have been taking care of me and my son during the many days I have been writing this thesis at their house. A loving thanks to my husband Aleksi, who has been my main supporter by encouraging me, commenting and advising with the text, giving me new ideas, and nursing our son, while I have been concentrating on this work. And a special thanks goes to my son Aaro for being the most important reason for me to finish this thesis.

Content

1. Introduction

During the last decades the trend in software business has been towards *global software development*. Geographically distributed development teams have become common in organisations. Virtual corporations, usage of subcontractors, and usage of team members from geographically distributed units of the same organisation are new cases in software development projects. Open source community has taken this trend to the maximum. Their idea is that individual programmers located anywhere in the world are connected to the Internet to read, redistribute, and modify the source code freely. Thus, the software evolves as people improve the code, adapt it and fix bugs.

Software configuration management (SCM) is one of the areas of software engineering. It is involved during the whole software project. Software configuration management controls the project by identifying the configuration of the system, recording and tracing changes to the system components, providing tools to control the changes and providing tools for auditing and reporting [Mordechai, 1994]. The members of a software project (especially developers) are involved with software configuration management functions in their everyday tasks. When the development of a project is divided into geographically different places, some of the above tasks get complicated.

The goals of global software development are to save time, save costs, shorten the time to market and share knowledge. These are the benefits that all organisations hope to gain, when starting global software development. However, there are also new challenges compared to the traditional local development. Global software development brings challenges in technical implementation of the development environment, in communication between people working in geographically different places, in handling the cultural differences of employees working around the world, and in the ways the organisations should work with virtual teams. Karolak [1998] has divided these challenges into three categories: organisational, communication and technical. I introduce a fourth category in this thesis: cultural. There are many research studies, which describe the impacts of these challenges in software projects [Carmel, 1999] [Damian and Zowghi, 2002] [Herbsleb *et al.*, 2001]. Asklund [1999] has presented models and architectures to implement global software development projects.

The intention of this thesis is to clarify how the known problems of global software development impact software configuration management systems. The impact of the different architectures and models is handled also. I have used a constructive research method to clarify the theories of software configuration management and global software development. As a result, I present viewpoints on issues that should be considered or resolved for software configuration management systems to manage the impacts of global software development. I also describe how some of these viewpoints could be solved in practise using one real global project as an example case.

This thesis is divided into six chapters. Chapter two presents history, concepts, methods, and tools of software configuration management and describes how software configuration management is used in software projects. Chapter three introduces the concept of global software development, why it has become a trend in software development, what benefits and challenges it has, and how it can be implemented. Chapter four sums up chapters two and three by describing how global software development impacts on software configuration management. Chapter five presents a global software project executed in Nokia Networks. The conclusions of the thesis are presented in Chapter six.

2. Software configuration management

Most people who have been participating in a software project have used some kind of a version control system. Version control is perhaps the most known and visible part of *software configuration management* (SCM), but software configuration management includes also other areas. Software configuration management is a discipline for controlling the whole evolution of software systems [Dart, 1991]. The role of software configuration management is to control the software project from the beginning to the end. Section 2.6 in this thesis will explain how SCM proceeds during the software lifecycle.

Projects usually have dedicated personnel responsible of software configuration management tasks. These people create and administer the software configuration system, train project members to use the software configuration management tool, write the SCM plan, mark the baselines of the software product, build the software product, and so on. However, there are many other SCM tasks, which are left for the developers, architects, testers, and other members of the project to do. A well-implemented SCM system is such that the software project members are not necessarily aware that they are doing software configuration management tasks. The concepts and disciplines of software configuration management described below usually blend with each other and with other tasks in a project, and they do not appear as such. SCM tasks are behind-the-scenes activities necessary to turn standalone software into a useful and usable commodity [Futrell *et al.*, 2002]. A task in which a programmer creates a new program file is a good example. When a new file is created and put into the version control system, the programmer does not necessarily need to know that a new configuration item is created with the help of configuration identification practices, and that it is stored with the help of configuration control practices, and that the status accounting functions are recording the whole process.

Software configuration management helps to deliver highly functional quality software, in time and to budget, and helps with the development, support, and maintenance tasks in the longer term [Thompson, 1997]. The purpose of SCM is to ensure that the software product is traceable and reproducible, and it helps managers and developers to ensure that the software product fulfils all of its requirements. Therefore, software configuration management is very closely connected to the product, the organisation, and the way they operate. Several authors and institutions have created well-defined disciplines and practises on how software configuration management should be managed. However, the way SCM is implemented in an organisation is deeply affected by the processes and disciplines organisations follow when producing software. On the other hand, if software configuration management is well defined, it has a great impact on overall software development processes of an organisation. The software configuration management disciplines are described in more detail in section 2.3.

Software configuration management has an effect on every phase of a software project like requirement management, design, implementation, testing and maintenance. The most important phase is the implementation, in which the function of software configuration management is to ensure a stable working environment for developers. SCM provides records of what has been done and by whom. This saves time, as developers do not need to be constantly in contact with each other to know what has happened in the development work. Software configuration management provides procedures that simplify the development process for the engineers, eliminate many sources of conflicts between project members, and institute logical change control process. Software configuration management offers valuable information for testing, quality assurance, project management, and maintenance. SCM can also be applied in planning budgets and staffing, writing specifications, and designing interfaces.

Berlack [1992] lists three reasons to consider software configuration management as an important function in software projects. First, SCM facilitates the ability to communicate the status of documents and implementation as well as changes that have been made. Second, corporate management looks at related software as an asset that can be used on other projects without the need to change or modify it. Third, software configuration management enhances the ability to provide maintenance support once the software is deployed in the field or sold in the marketplace. SCM does so through well-identified software elements and a history of the development of software, which enable a cost-effective fix with little impact on user or customer. These reasons concentrate on implementation and maintenance phase of software process and also on reuse of software components. On a more abstract level, Kelly [1996] has presented five criteria in which software configuration management is meant to ensure that

- you know what you have got to produce,
- once you have got it, you know where it is and what state it is in,
- only the right people can use or change it and they will understand the impact of that change,
- useful reports are available and

- the agreed procedures are being followed, so that everything hangs together properly.

The overall goal of software configuration management is to maximise productivity by minimising mistakes.

Software configuration management can be divided into different methods and procedures. There are methods to control different versions of different components, methods to control configurations and their versions, and procedures for creating and modifying versions and configurations. In the maintenance phase of a project, software configuration management clarifies the configurations each customer has, the compatibility information of components' versions, the parts impacted by planned changes, and the information needed for rebuilding the version of the product of a certain customer. All the above are examples of software configuration management tasks at a more detailed level.

2.1. Background

Configuration management got its start in the defence industry environment after World War II as a management technique and a discipline to resolve problems of poor quality, wrong parts ordered and parts not fitting [Berlack, 1992]. The need for a discipline to identify and control the design and to communicate information was most apparent in the defence industry, where the expected high-quality workmanship appeared to be slipping. The first standard for configuration management was authored and published in 1962 by the US Air Force. In 1968, first instructions were published that divided configuration management to description and definition of the major components and activities, change control, specifications, and status accounting. In 1971, a first standard was published that recognised also configuration management of software. After this standard many standards of software configuration management were written. The first approved standard was DOD STD 2167. It divided configuration management using the phases of the life cycle of a project [Berlack, 1992]. Within each phase, it described the activities to be performed, the product expected from these activities, the design reviews that were required for approvals, and the role of software configuration management in capturing the documented descriptions and subsequent change paper. After the first approved software configuration management standard, many standards and guides for SCM have been published. For example, the following organisations have published SCM standards: Electronic Industries Association (EIA), Institute of Electrical and Electronics Engineers Inc. (IEEE), Society of Automotive Engineers (SAE), American National Standards Institute (ANSI), and International Standardisation Organisation (ISO).

Today there are several definitions of software configuration management. Mordechai [1994] defines that configuration management is a management discipline, which

- identifies the proposed or implemented configuration of a system at discrete points in time,
- systematically records and traces changes to all system components,
- provides tools for controlling changes, and finally,
- allows everything happening with the system, throughout the entire life cycle of the system, to be verified via auditing and reporting tools.

This definition quite well points out the four main areas of software configuration management: identification, configuration control, status accounting, and auditing. There is a section of each one of these areas in section 2.3. All these areas are important to guarantee integrity, accountability, visibility, reproducibility, project coordination, traceability, and formal control of product evolution [Mordechai, 1994]. The four main areas are key issues when trying to improve the effectiveness of projects and helping in maintenance tasks.

2.2. Main concepts

Software products consist of different *components* such as program files and documents. Binaries and other derived objects, which are produced from other components by some automatic method, are also components. The term component is used to describe an identifiable part of a project [Kelly, 1996]. Software components may be made up of several modules, with each component itself forming a part of the whole system.

Configurations are collections of components that form a product or a part of a product. They can be thought of as functional units that are defined in technical documentation and achieved in a product. One configuration can include other configurations. New configurations are normally created when new versions of components appear. Often a reason to form a new configuration is that customer-specific changes are needed or different hardware or software environment is taken into use.

Figure 1 shows how configurations are created. One version of each component is selected to a configuration. Configuration A includes the first version of Component 1, the second version of Component 2, and the second version of Component 3. Configuration B includes the third version of

Component 1, the same second version of Component 2 as Configuration A and the third version of Component 3. It could be that the Configuration B is created to replace Configuration A, because of the new versions of Component 1 and Component 3.

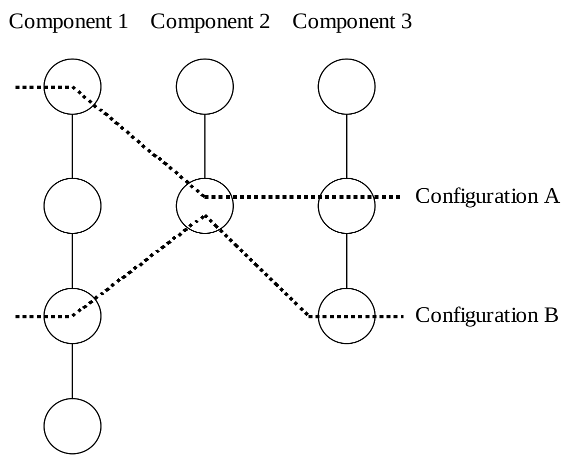

Figure 1. A configuration is a collection of components.

Both components and configurations are called *configuration items* (CI) in software configuration management systems. Formally a configuration item is a part of the system that needs to be independently identified, stored, tested, reviewed, used, changed, delivered or maintained during development or delivery [Kelly, 1996]. Configuration items are usually identified by mnemonics and some kind of a running numbering system. Configuration items can vary in complexity, size and type.

The first *version* of a configuration item is created, when the configuration item is put under control of software configuration management. This also means that the configuration item is *frozen*. Freezing a configuration item means that no one can modify that version of the configuration item anymore. When the configuration item needs modifications, a new copy is created from the frozen version. Then the copy is modified and frozen as a new version.

There are two types of versions: *revisions* and *variations*. New versions that supersede old version are called revisions. They represent the evolution in a software project, when bugs are corrected or new functionalities are added, and the intention is to make the older versions obsolete. Variation is an equal alternative of one version. Variation fulfils the same function, but for a slightly different situation. An example of a different situation is a different hardware environment. The set of all revisions and variations that belong to one configuration item is called a *version tree*. *Branch* is a variant development path in a version tree. Variations can be *merged*. The two variations are compared to each other and to their common ancestor to perform the merge.

Figure 2 shows how configurations are formed using specific versions of components or other configurations. Circles in the figure represent versions of the illustrated configuration items. All versions are identified using a running numbering system. An arrow from a circle to a box of configuration items signifies that the version is created using the configuration items inside the box. The selected versions are filled with black colour. Feature1, feature2, and feature3 are components and subproductA, subproductB, and productX are configurations. Version 4 of feature1, version 3 of feature2 and version 3.1.1 of feature3 are used to create version 2.2.2 of subproductB. Versions 2 of subproductA and version 2.2.2 of subproductB are used to create version 2 of productX.

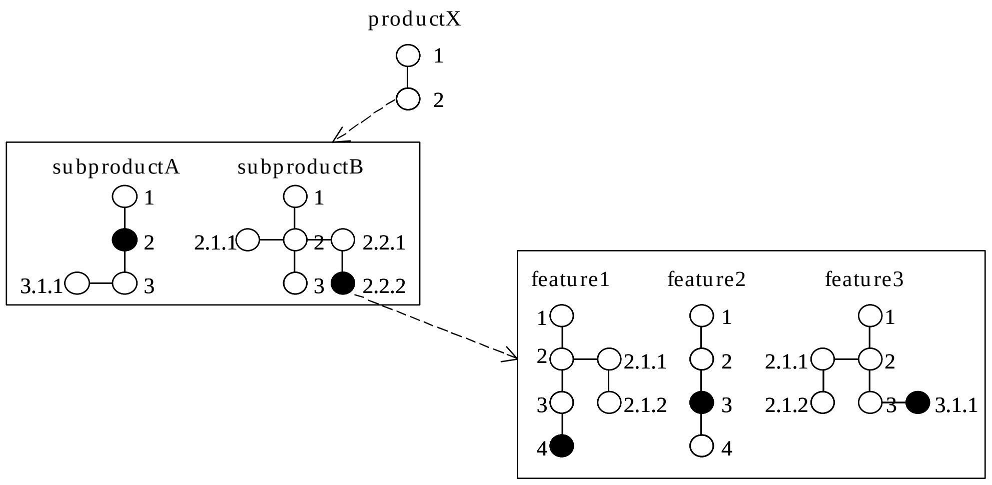

Figure 2. Configurations and version trees.

A baseline is a specification or a product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures [IEEE, 1990]. Before a baseline is created, the developer can make changes to the configuration items unofficially upon her consideration. Baseline is a reference point in the life cycle of a project and it relates to project milestones, as defined in the quality plan of the project. There are three different kinds of baselines: *functional* baselines, *allocated* baselines, and *product* baselines. The functional baseline is the initially approved technical documentation describing the functional characteristics of an item, and the verification required to demonstrate the achievement of those specified functional characteristics. The allocated baseline is the initially approved specification governing the development of configuration items that are a part of a higher-level configuration item. The specification includes description of the functional and interface characteristics of an item. The product baseline is the initially approved technical documentation defining a configuration item during the production, operation, maintenance and logistic support of its life cycle.

System *building* is the process of combining source components of a system into components, which execute on a particular target configuration [Leon, 2000]. A build is defined as an operation that takes one or more configuration items and performs some action on them to create new deliverable configuration item [Thompson, 1997]. It is the compilation and integration process. At the time of the build, it is likely that several versions of the configuration items exist, so the selection of the correct configuration items is critical to success. When the source components are changing, the system or parts of it have to be rebuilt. *Release* includes the executable code, installation files, data files, set-up programs, and electronic and paper documents [Leon, 2000]. A system release is the set of items that is given to the customer. Each system release includes new functionality or feature, or some fixes for the faults found by customers, developers or testers.

Traceability means that a complete history of all configuration items is known and can be proven. It also means the information about which program configurations a component has been included in. Part of *repeatability* is the ability to reproduce a configuration item, baseline or total configuration exactly as it was at a given point in time or in a given release. Repeatability also ensures the possibility to verify that the reproduction has been correctly implemented.

2.3. Different areas of software configuration management

There are some variations in the literature how the main areas of software configuration management have been defined. IEEE Guide to Software Configuration Management [1987] lists following as primary activities: *configuration identification*, *configuration control*, *status accounting* and *auditing*. The division is represented in Figure 3. The four primary activities are practised throughout the project and the next sections describe them more.

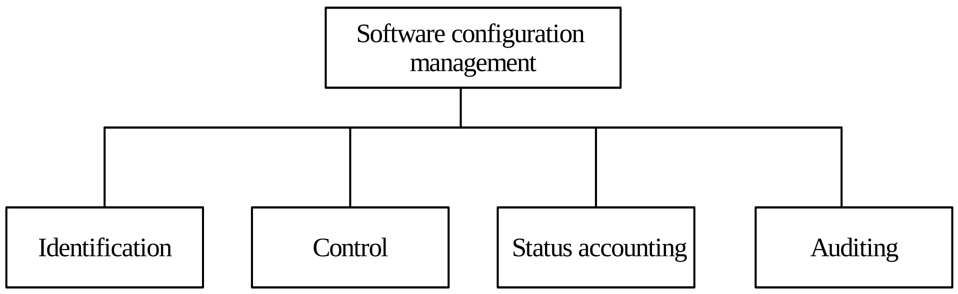

Figure 3. SCM divides into four primary activities.

Sometimes also interface control, subcontractor control, process management, and teamwork are considered as independent activities of software configuration management [Berlack, 1992] [Dart, 1991]. Interface control is about the evaluation, coordination, and approval or disapproval of all proposed changes to established functional and physical interfaces as defined in specifications, documents, and drawings. Subcontractor control is about the evaluations, coordination, and approval or disapproval of all changes submitted by the subcontractor to approved configuration documentation. It is also about the monitoring of the subcontractor's performance of the software configuration management function. Process management ensures that all procedures, policies, and lifecycle model of the organisation are followed. Teamwork controls the work and interactions between multiple users on a product.

Before configuration identification can start, the following phases in software configuration management need to be implemented [Leon, 2000]. First the software configuration management system must be designed. If the company already practices and has guidelines on SCM, this phase is easy. Nevertheless, because no two projects are the same, the guidelines need to be customised to suit the needs of the current project. In addition to the guideline customisation, questions on which software configuration management tool to use and how, are to be decided in software configuration management system design phase. After the initial system design has been done, a software configuration management plan is to be written and the SCM team will be organised. The software configuration management team size can vary from a single person to a full-fledged team depending on the project. Training of the team members and the project members on how the software configuration management is practised in the project is the last step before the project members can start the identification of their configuration items.

2.3.1. Configuration identification

Configuration identification is the most essential part of software configuration management. If nothing is identified, nothing can be controlled. The official definition of IEEE [1990] says that configuration identification is an element of configuration management, consisting of selecting the configuration items for a system and recording their functional and physical characteristics in technical documentation. Identification defines in higher level what belongs to the total configuration and plans an identification method for configuration items, baselines and the whole product. Planning the identification scheme is an important activity, because the scheme is used throughout the life cycle of the software product and because the structure of the product is not yet completely known.

An identification method for configuration items includes planning the structures of all configuration items of the system and the relationships between the configuration items. Poor selection of configuration items can affect costs and scheduling, and can become an unnecessary administrative and technical burden [Leon, 2000]. An identification method for configuration items also includes planning naming conventions and labelling of all the actual configuration items, their files, and releases. Naming conventions must uniquely identify each configuration item and they can use the hierarchy of designed items to make the identification effective [Whitgift, 1991]. A good naming system will make it possible to understand the relationships between the configuration items from their names. The decision on the complexity and detail of the configuration item names depends on the size and complexity of the project.

Labels are attached to the configuration items when the items need to be marked for some purpose. Most common cases are when configuration items are attached to some larger configuration. The labelling can happen during a build or when marking a baseline, for example. During the labelling process labels and date and time stamps are attached to all involved configuration items. Labels contain a numbering system or mnemonics or both. Numbering systems usually consist of two parts that are separated by a dot. The number on the left of the dot denotes the number of the last baseline and the number on the right of the dot denotes the current revision from the last baseline [Mordechai, 1994]. Generally, the second part will be zeroed when new baseline occurs. Mnemonics can be given to each configuration item and build to accompany the numbering system. Mnemonics can be derived from the project or system and the type of the item and the item name. An example of a label is SS_TOOLS_2.12. This is a label of a subsystem (SS) of some larger software product. This subsystem is meant to produce tools for controlling the software product. Number two indicates that the second baseline of the project has been created. With this label we are labelling the 12th build after the creation of the second baseline.

An identification method for baselines involves planning when the baselines will occur. Baselines are connected to a life cycle of a software project and they can mark the end of one phase or segment of a phase, or a beginning of a new phase. Therefore the criteria for every baseline must be decided in an early stage of project and it must be followed from the beginning to the end. Changing criteria can cause enormous amount of work. Baselines define the state of the system at a given point of time [Mordechai, 1994]. So, it is obvious that recording this information can be critical for the success of the project. After the first baseline, changes to the approved versions of configuration items must be made through a formal review and agreement.

An identification method for the product is mainly planning the structure of the whole software hierarchy. This is the first time when a structural overview of the software system and its items is presented. When the structure is planned early enough it is easier to preassign document numbers, keep track of the progress as it matures, and estimate the manpower and resources that will be required. It also helps to select the candidate configuration items. The chosen identification method should reflect the structure of the product, the project, and the organisation and it should ensure visibility and traceability of the software. Visibility permits the software to be seen by anyone who is allowed to see it. Traceability is the ability to link individual events and parts to each other in time.

2.3.2. Configuration control

When all items are identified, there comes a need to control them. Configuration control (also known as *change management and control*) does that by controlling the database where all configuration items are stored, providing ways to report problems in the system, and controlling changes that are made to the configuration items. Configuration control provides methods to control the system implementation process. The goal is to manage the life cycle of the software in a controlled way, so that nothing unexpected or unplanned could happen. Configuration control is the answer to the following most common problems the developers have: shared data, double maintenance and parallel updates. Configuration control is the software configuration management function that is performed most often. The activities of configuration control will increase as the project evolves, because more and more items will undergo change, more and more people will be inducted, requirements will change, new modules and subsystems will be added, different versions will have to be maintained, and so on.

The database where all configuration items are safely stored is called *controlled area* (or library). It contains all items that are essential to the project: source code, user and system documentation, test data, specifications, project plans, and derived items. Controlled area uses access control to safeguard items. Access control governs which developers can access and modify which configuration items [Pressman, 1997]. It takes care that only right people can have access to right versions of right configuration items to browse and modify them. Synchronisation control helps to ensure that parallel changes made by two different people do not overwrite one another [Pressman, 1997]. Access control and synchronisation control can be handled automatically by version control systems (like RCS [Tichy, 1985] or SCCS [Rochkind, 1975]). These systems usually use checkout-checkin model, in which user first has to check the wanted item out of the database to modify it and the item is checked back in when modifications are done. Only people who have access rights to modify items can check them out and, while item is checked out, nobody else can modify the same version of it.

Problem reporting is the mechanism to report problems that appear in configuration items. There are two classes of problems to which the activity of problem reporting is based on: errors reported and anomalies discovered [Mordechai, 1994]. Error is a mistake in the code, in the design or in the requirements specification, and anomaly is unintended behaviour in the software, for example. Problem reporting includes investigating the problems and their satisfactory clearance. The consequence of problem reporting can be cancelling the report, creating a new configuration item or modifying an existing configuration item [Kelly, 1996]. Problem reporting guides what should be the content of the report. The report should include the full identity of the item exhibiting the problem, the nature of the problem, the circumstances in which the problem occurred, the environment in which the problem occurred, diagnostic information, and the effect of the problem [Whitgift, 1991].

The *change control* process starts from a *change request*. A change request can be based on a problem report, or an enhancement idea, or a new product feature. The change requests are classified into different categories with different priorities. They are evaluated and analysed in terms of impact to system functionality, interfaces, utility, cost, schedule, software safety, reliability, maintainability, efficiency, and so on. An appropriate authority will approve, disapprove or defer the change depending on the criticality of the change request. Usually major changes need the approval of a *Change Control Board* (CCB). A named individual (a configuration management officer) or a member of the software configuration management team can approve the minor changes [Leon, 2000]. The composition of a Change Control Board can vary from a single person to a highly structured and very formal set-up with many people, depending on the complexity, size, and nature of the project. In large projects, the CCB should contain a representative from the software configuration management group, and representatives from the project team, quality assurance group, company management, and marketing. After the approval, the change can be implemented and verified at the system level. A change history is recording the events that occurred to items from the state before change to the one after.

The ability to make changes rapidly is a big benefit of the existence of software [Mordechai, 1994]. It is one of the main reasons why software is everywhere. Changes to the requirements drive the design, and design changes affect the code [Paulk *et al.*, 1995]. Testing uncovers problems that result in further changes, sometime even in the original requirement. At the same time, the intangible nature and susceptibility to change make software difficult to control [Thompson, 1997]. In the software configuration management point of view, changes can be quite uncontrolled and developers can do changes as much as they want by themselves until the first baseline occurs. After that, changes have to be controlled. Uncontrolled changes lead quickly to chaos and they slow down the project. Often changes are documented, but the impact of the changes is not analysed. That is why a change request should be made and evaluated before the change is made. After a thorough impact analysis for both technical and resources or timescales have been done, a configuration item can be updated [Kelly, 1996]. One part of the update is to ensure that the changed configuration item is reviewed and tested to ensure that the whole approved change, and nothing else, is implemented.

2.3.3. Status accounting

IEEE [1990] defines configuration status accounting as an element of configuration management, consisting of the record and reporting of information needed to manage a configuration effectively. This information includes a listing of the approved configuration identification, the status of proposed changes to the configuration, and the implementation status of approved changes. Status accounting makes configuration management visible by recording the status of all items and change requests. It is the information gathering and dissemination component of software configuration management [Leon, 2000]. It is a collection of information that provides answers to questions like what happened, who did it, when it happened, what were the reasons, and what was affected.

Status accounting is the element of software configuration management that records information on different activities. These are creation of a new configuration item, update of an existing configuration item, and approval of a change, for example. Different people need this information in different forms and in different phases of the project. That should be considered when planning status accounting. Only right information should be recorded at the right time so that correct reports could be made when they are needed. At minimum, status accounting reports the transactions occurring between SCM-controlled entities [IEEE, 1987]. Other important reports are change logs and delta reports [Mordechai, 1994]. Transaction reports show the effect and relationships resulting from each event that has occurred during the project. Change logs contain all information of requested changes. Delta reports summarise the progress of the development and compare it to the progress presented in the previous report. Also resource usage, status of all configuration items, change in process, and progress reports are typical reports [Mordechai, 1994; Leon, 2000].

The output of status accounting must be accessible to all members of the project. The project members can follow how the project is progressing compared to the project plan using reports produced by status accounting. They can check forthcoming changes that can have an affect on their configuration items. They can also use status accounting information to see the history of an item. The purpose of status accounting is to communicate information to the project, users or support activities as soon as it becomes available. Thus, status accounting plays a significant role in the success of software projects. The status accounting reports are invaluable also during the maintenance phase. To understand and identify the cause of a problem, it is important for the maintenance staff to know the history of the configuration items. The recorded data must be in a form, which allows traceability from top to bottom and bottom to top for software in development, in the field or sold in the open market [Berlack, 1992]. The history records can also be used to analyse and improve the software development process.

2.3.4. Auditing

Configuration auditing ensures that all procedures are followed correctly and that the information of configuration items and their structures is correct. IEEE [1990] divides configuration audits to *functional* and *physical* audits. Functional audit is an audit conducted to verify that the development of a configuration item has been completed satisfactorily, that the item has achieved the performance and functional characteristics specified in the configuration identification, and that its operational and support documents are complete and satisfactory. Functional audits normally involve a well-defined sequence of tests designed to ensure that the performance of the item conforms to the requirements in the specification. The process may include some or all of the following forms of tests, analysis or demonstrations: environmental tests, reliability tests, user trials, interfaces with other systems, software testing, and stress testing [Leon, 2000]. Physical audit is an audit conducted to verify that a configuration item conforms to the technical documentation that defines it. When a physical audit is completed, the product baseline is established. Software configuration management auditing completes the technical review by evaluating the characteristics of the configuration item that are not commonly considered during the review.

Quality assurance performs audits to control change procedures and other activities, but it has other criteria to determine product integrity and reliability. Software configuration management audits concentrate on verifying that a software product is a consistent and well-defined collection of parts. Configuration audit of the developed software product provides assurance that what was required has been "built" as evidenced by the software test reports, documentation, and media [Berlack, 1992]. At a minimum, the configuration should be audited when the product baseline is established and whenever it is subsequently changed due to the release of a new version of the software. Auditing the final release gives the company and the customer the satisfaction of knowing that what they are delivering or getting is complete and meets the requirements. Leon [2000] recommends an external auditor to do the configuration audit because the auditing activity requires a very high degree of objectivity and professionalism. It is the responsibility of the SCM team to schedule the audits and find qualified personnel to perform them. The person who conducts the audit should be knowledgeable about SCM activities and functions, and technically competent to understand the functionality of the project.

2.4. Software configuration management plan

Once the software configuration management system is designed, it should be documented. The document should make the working of the SCM system, the procedures, and the functions, duties, and responsibilities of each member transparent and known to all members of the software configuration management team, project team, the possible subcontractor team, and others [Leon, 2000]. This document is called *the software configuration management plan* (SCM plan). Earlier studies have examined the key elements of a successful software configuration management solution. They all agree that the SCM plan is one of them. Bounds and Dart [1993] present three such key elements. In addition to SCM plan, they introduced the software configuration management system and the software configuration management adoption strategy. Moreira [1999] listed four key elements: SCM plan, skilled SCM personnel, funding, and sponsorship. By funding Moreira means money to purchase appropriate tools and infrastructure, and by sponsorship he means commitment of management to the effort. Capability maturity model [Paulk *et al.*, 1995] defines software configuration management plan as the first step in establishing a software configuration management system. The SCM plan creates awareness among project team members, software configuration management team members, and other people who are in some way related to the project. It documents what and how the software configuration management activities are to be done, who is responsible for doing specific activities, when they are to happen, and what resources are required [IEEE, 1998]. In addition, the SCM plan forms the basis of training the personnel who are a part of the project team or the software configuration management team. It will also be used in the resolution of conflicts regarding the practise or implementation of software configuration management functions in the project [Leon, 2000].

There are several standards written [IEEE, 1998; MIL, 1994; ISO, 1995] for the software configuration management plan. Bounds and Dart [1993] found out in their survey that standards prove invaluable in assisting a person in writing SCM plan, since they provide the basic framework, and act as a guideline for writing the plan. The format specified by most of the standards is similar and they offer considerable latitude and freedom to the person who writes the plan [Leon, 2000]. All standards expect the plan author to define some topics such as scope, purpose, definitions, software configuration management organisation, software configuration management functions, responsibilities, and resources. The degree of detail and amount of additional information, as well as the format of the information, depends on the writer and the nature of the project.

Leon [2000] has presented a sample outline of a software configuration management plan. This outline is shortly introduced here as an example structure. The plan contains six chapters: introduction, SCM management, SCM activities, SCM schedules, SCM resources, and SCM plan maintenance. *Introduction* provides an overview of the plan: purpose, scope, definitions and references. It should give the user a clearer understanding of the plan. *SCM management* gives information about the software configuration management organisation, software configuration management responsibilities, relationship of software configuration management to the software process life cycle, interfaces to other organisations in the project, and software configuration management responsibilities of the organisations. It describes the organisational structure of the software configuration management team, the duties and responsibilities of all those involved in carrying out the software configuration management activities, how the SCM team will interact with other organisations in the project, and the responsibilities of the vendors, subcontractors and other organisations in relation to the carrying out of software configuration management functions. It also relates the software configuration management activities to the different phases of the software development life cycle. *SCM activities* concentrates on the four primary activities of software configuration management with addition of interface control and subcontractor control. The first part of the chapter discusses about configuration identification. It identifies the items to be selected as configuration items, specifies the identification system, describes how the configuration items are to be stored, and how the access to them will be controlled. The second part concentrates on configuration control. It describes how to initiate a change, how the evaluation of a change request is carried out, how the change request is processed, and how the approved change request is to be implemented. It also describes the functioning of the Change Control Board. The third part of the SCM activities chapter covers configuration status accounting. It describes the information requirements of the project, how the status accounting information is gathered, the various reports that will be created, how and to whom the status accounting information will be disseminated, and detailed information about the releases. The fourth part of the chapter discusses about configuration auditing. It describes the different types of audits that will be performed, the procedure to be followed for each audit, and the activities that should be carried out after the audit. It also specifies the list of configuration items that are to be audited. *SCM schedules* chapter describes the sequence of the software configuration management activities, their interdependencies and relationship to the project life cycle and project milestones. This chapter of the plan also establishes the schedule for the different configuration audits. *SCM resources* chapter identifies the software tools, techniques, equipment, personnel, and training necessary for the implementation of the software configuration management activities. *SCM plan maintenance* describes the activities that are required to keep the plan current during the life cycle of the project.

2.5. Tools

During the software configuration management system design phase the selection of the used software configuration management tool has to be made. Every software development team has some SCM tools and methodologies [Leblang and Levine, 1995]. In the simplest case, developers send email, talk to each other on the corridors, and tape notes to their monitors. Some developers working with UNIX depend on free UNIX utilities (like RCS and Make) layered with custom scripts to implement procedures for keeping track of what has changed. These systems were common in 1980s and focussed closely on file control [Estublier, 2000]. Some organisations have built up scripts to create developer “sandboxes” that mirror the contents and structure of predetermined baseline. The examples above can work in small teams and organisations. But the software configuration management demands grow as the organisation hires more people, supports a longer history, and accumulates more code.

A software configuration management tool should help the project team to manage all the main areas of SCM. Dart [1991] defines following as the overall functional requirements:

- identifies, classifies, stores, and accesses the components of the software product,
- represents the architecture of the product,
- supports the construction of the product and its artefacts,
- keeps an audit trail of the product and its processes,
- gathers statistics about the product and the process,
- controls how and when changes are made,
- supports the management of how the product evolves, and
- enables a project team to develop and maintain a family of products.

The first generation of software configuration management systems did not meet these requirements, but the second-generation products largely satisfy the functional requirements [Hoek *et al.*, 1995]. The first generation software configuration management tools, such as RCS and SCCS, focus mainly on version and release control. The second-generation tools, such as DSEE [Leblang and Chase, 1987], pay much more attention to network support for parallel software development [Chan and Hung, 1997].

The software configuration management tools commercially available today are full-fledged tools that offer such diverse features as build management, defect and enhancement tracking, requirement tracking, release management, software production control, software packaging and distribution control, and site management. They provide automated support for maintaining control over the evolution of a software system by structuring the work of developers, providing visibility into the work of others, and gathering all the system’s components together [Grinter, 1996]. The market leaders currently are Continuus and ClearCase [Estublier, 2000]. Referring to the above these tools can be called the third generation software configuration management tools. The third generation tools reduce development time by reducing mistakes, tracking problems and rebuilding systems easily and quickly. They automate the most of the monotonous and repetitive tasks that were earlier done by people. These tools have all the information programmers, managers, analysts, auditors or any other people in the project need, and they can deliver the information to the users in any format almost instantly [Leon, 2000]. The second half of 1990s saw the consecration of software configuration management, as a mature, reliable, and essential technology for successful software development, and many observers consider SCM as one of the very few software engineering successes [Estublier, 2000]. But as Moreira [1999] listed sponsorship as one of the key elements in successful SCM implementation also Grinter [1996] has noticed in her empirical studies that the usage of SCM tools depends on the surrounding organisational and social context.

Leblang and Levine defined in 1995 that the challenges for a good SCM tool are *scaling*, *product complexity* and *importance of history* [Leblang and Levine, 1995]. A few years later, Estublier [2000] added *interoperability with the other software engineering tools* and *efficiency* to the list. These challenges are still valid and at least multi-user support, graphical user interface, ease of set-up and process management can be added. Most software development organisations consist of tens (or hundreds) of engineers, tens of thousands of source files, and millions of lines of code - all scattered across dozens of machines. So for this large-scale programming environment, software configuration management must encourage parallel development by combining flexibility with absolute safety. A large-scale software development organisation produces dozens of applications and all from different combinations of the same underlying source code. Thus, in spite of this complexity, a developer must be able to select the initial set of file versions that make sense for a particular project, and evolve the software from that point. The importance of history emerges when organisation needs to reliably reproduce any software build or do a bug fix or a minor enhancement to an old release, for example.

2.6. Software configuration management in a lifecycle of a project

Software configuration management communicates with all of the software project activities. SCM collects their outputs and products. Its role starts with a product proposal and continues through the product release to the customer or when it is turned over to a support facility [Berlack, 1992]. Figure 4 describes what activities of software configuration management are involved in different phases of a software project. In the figure, the software project is divided into six phases: concepts, requirements analysis, design, implementation, testing, and delivery. The inputs for the project come from the contract made with the customer. The requirements for software configuration management are also indicated in the contract. It is presumed that the initial software configuration management system is already designed and this process describes a project specific SCM system.

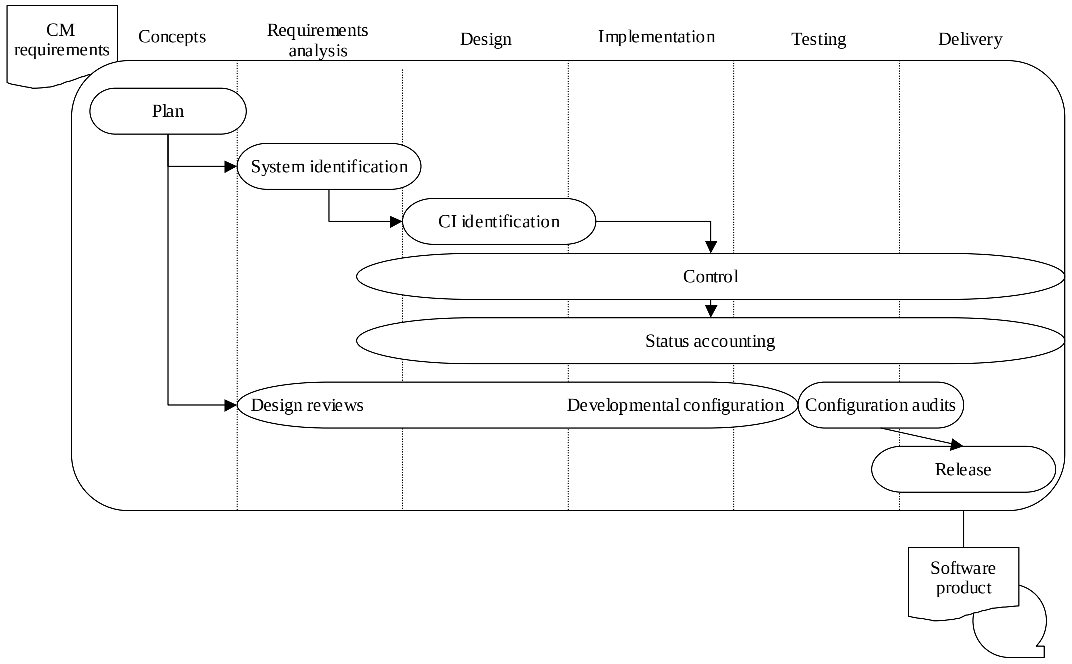

Figure 4. SCM process model modified from the original picture of H. Roland Berlack [1992].

In Figure 4, the initial activity of software configuration management process is the software configuration management plan. SCM plan defines and documents the software configuration management concepts at the same time as the concepts of the whole software project are defined. The next phase of software project in Figure 4 is requirements analysis, which starts after concepts are defined. At the same time, software configuration management process moves on to system identification. The initial software configuration management system is customised for this project according to the procedures documented in the SCM plan. Configuration control and status accounting activities are started at the end of requirements analysis phase. These activities are needed when design phase of the software process starts and first configuration items are identified.

All configuration items are identified during the design phase of the SCM process model in Figure 4. The planned milestones and the structure of the software product have impact on the configuration identification activities. Control and status accounting activities are ongoing through implementation, testing, and delivery phases of software process. These activities provide the means of communicating any changes that have been made to the configuration items. The software configuration management plan is the input for design reviews and developmental configuration, which prevail throughout the development cycle. Configuration audits take place at the end of the testing phase. The audits ensure that the configuration items are correct and complete. A release is produced as the last software configuration management activity in Figure 4 during the delivery phase of the software process.

Software configuration management provides security, control and status accounting software project members. A good example is quality assurance, which tries to ensure the integrity of the software product from beginning to end. Quality assurance monitors the performance of other activities including software configuration management. Software configuration management provides status accounting activities to quality assurance. Software configuration management communicates closely also with maintainer function and data management. Data management and software configuration management relate in terms of the specifications and documents. Software configuration management ensures correct identification, records change history, and maintains status information for specifications and other documents as they change or are released. SCM provides information to maintainer function about the software product and process changes, and initiating releases of revised software.

Software configuration management should be in place at the start of the project to communicate with all parts of project organisation. This includes activities like the identification and approval of documents, changes that have occurred or are pending, and releases, deliveries and returns. The role of software configuration management in the software project starts with the response to a request for a bid, estimate or proposal and ends by initiating action to turn over products of the development phase to the customer or support facility. Along with software engineering, software configuration management relies a great deal on the methodology, analysis and trades carried out by system engineering in determining the levels of functional performance required to design, develop, build, and test the software product [Berlack, 1992]. Once software configuration management has the knowledge of the software hierarchy, its role is to begin planning for the amount of documents and changes that will occur. When software engineering has determined the number of lines of source code, the number of changes can be estimated based on previous history.

Do [1999] has described how software configuration management is proceeding during software lifecycle. In this example, the software lifecycle is divided into phases according to the waterfall model. In the system requirement analysis phase, a software configuration management plan should be written and detailed procedures should be prepared. Also SCM tools should be chosen. When system requirements and system design documents are reviewed, a software functional baseline is established and all documents are placed under configuration control. Software requirements analysis identifies all requirements towards the software to be developed as configuration items. Once configuration items are approved, they are placed under software configuration management control and can be considered the allocated baseline. After this baseline a formal change recommendation must be prepared and submitted for approval for every change impacting a software configuration item. During the design and coding phase software configuration management can determine the impact to subsequent modules when changes to specific modules are requested. After the code is tested, detected software anomalies are analysed to determine the causes. Proposed change impacting any configuration item is reviewed in the Change Control Board meeting to determine severity, priority, and cost and schedule impact. This SCM function is crucial to assure product quality. In the final phase of the software development cycle, configuration audits control the software delivery to the customer. In the functional configuration audit, formal tests are conducted to verify that configuration items meet all software requirement specifications. The physical configuration audit allows verifying and validating that the software release is documented in the version description document, with software modules, test procedures, and results that prove the required functionality. This documentation becomes a product baseline. It is placed under configuration control and delivered to the customer.

2.7. Summary

There have been standards and guides for software configuration management for about 30 years. Different organisations define software configuration management slightly differently, but the main ideas are the same. Software configuration management is about identifying and controlling the software. There are different ways to do it and different tools to use for help, but the goal is always the same: get software projects ready in time and with best possible quality.

To achieve the goals in software business, organisations are utilising the four main disciplines of software configuration management: configuration identification, configuration control, status accounting, and auditing. Configuration identification includes methods for identifying configuration items, baselines, and the whole product. Configuration control controls the configuration item database, changes made to the configuration items and the problem reports. Status accounting takes care of recording the status of all items and change requests. Auditing ensures that all procedures are followed and all information is correct.

Software configuration management has a significant role in software project since it is involved in all phases of a project. It is used when first contracts are made to start a new project and it is still involved when project has been transferred to the maintenance phase. It should be recognised at a high priority in the organisation to have visibility over the design, development and testing [Do, 1999]. An effective SCM system will help to improve productivity of the staff and decrease ramp up time for new employees. Software configuration management communicates with many other project activities, such as quality assurance and data management. However, the main function of SCM is to help development and ensure that the work of developers is safe and controlled.

Though software configuration management has been used in software projects over a few decades, I have noticed that organisations did not invest on it very much in the near past. Organisations understood that version control and change control are helping development, but they did not always concentrate on the other areas of SCM. Nowadays, when software projects are getting larger and market demands are getting higher, software companies have understood the importance of software configuration management. To survive and stay competitive in the market, software manufactures must eliminate inefficiencies in their software development lifecycle and minimize the time it takes to revise their products. The ability to manage change effectively is a major key to success [Do, 1999]. More complex software projects are also bringing new demands on software configuration management. In the early 80's, software configuration management focused in *programming in the large* (such as versioning) and in the 90s in *programming in the many* (such as concurrent engineering) [Estublier, 2000]. Late 90s the focus turned to *programming in the wide,* which brought out the demand of global software development.

3. Global software development

In *global software development* (GSD), the software development activities are distributed across multiple sites [Mockus and Herbsleb, 2001]. These sites can be located anywhere. In a smallest case, there are two sites in the same city. In a large project, there can be many sites in several different countries around the world. In all of these sites, there are teams working on a common software project. They are working independently at some level, but sharing a common software base.

There are several reasons why global software development has emerged. Carmel and Agarwal [2001] state that today two critical, strategic reasons for global software development are cost advantages and a large labour pool. Organisations are seeking lower costs and access to skilled resources from remotely located facilities or using outsourcing. The problem with the resources is that the traditional organisations are situated on areas, where local resource pool is limited. In addition, in the mid 1990s the labour costs escalated as companies competed for resources [Karolak, 1998]. However, there are centres of software R&D growing outside the traditional centres (such as USA) [Carmel and Agarwal, 2001]. In these emerging centres, cost savings can be achieved from low software labour costs. Also using the emerging centres along with the traditional centres offers a possibility for organisations to utilise the round-the-clock development. The new centres provide necessary experts with cheaper costs who are willing to do the less exiting tasks such as maintenance, porting and testing [Carmel, 1999]. For western European organisations this means collaboration with organisations in Russia or Hungary, for example. To also benefit from round-the-clock development a possible choice for a western European organisation is to expand in China, India or Philippines.

Saving costs is the main benefit the organisations are looking for in global software development. The new locations demand some investments in premises, communication lines, and computers, but those are insignificant compared to the savings. But as global software development becomes more commonplace, there is a possibility for organisations to consider the individual work preferences. Working at home or working on the road can have positive effect on those employees that are able and willing to use it.

3.1. Background

Global software development became more common at the last decade. In the early 1990’s, the number of entities engaging global software development was small, but this has rapidly changed [Carmel and Agarwal, 2001]. MacKay [1995] says that geographically distributed development is simply recognition of the true state of software development today. More than 50 nations are currently participating in collaborative software development internationally. Leon [2000] states that as communications and information technology make rapid strides forward, distributed development is going to be commonplace.

In 1990's, software industry became under renewed pressure to achieve higher software quality and better customer support. There was also a pressure to increase productivity, reduce development time, and increase reuse of software components [Murugesan, 1999]. Companies had to think how to enhance the efficiency of their organisations. Many large companies had employees in different cities and countries working on separate projects. These companies started to create distributed teams to work on the same software projects [Whitehead, 1999]. Another driver in the past was the need to be locally present for customisation and after-sales service [Ebert and De Neve, 2001]. Showing local customers how many new local jobs were created could justify more contracts. Also cross-organisational projects started to occur more frequently, such as using a subcontractor. Finally, the most challenging application areas demanded the formation of virtual corporations in order to bring together the necessary key competence and resources [Cocchio and Puttero, 1999].

For a long time, organisations have had access to global networks. Earlier it meant that some programmers could connect their terminals to the central mainframe through telephone or other slow communication lines. However, the PC revolution and the rapid development of the Internet in 1990’s brought a dramatically increased access to global networks [Carmel, 1999] [Asklund, 1999]. What happened in the 1990's was that the underlying networking technology was developing quickly and workstations and personal computers started to dominate the workplaces. The network bandwidth was increasing, the costs were decreasing, and the microcomputers and controllers gain better price-performance ratio. Thus, the emergence of the global network made it possible to remotely access information across organisational and national boundaries [Haag *et al.*, 1997].

The Internet and the World Wide Web have facilitated global software development as a new model of software development [Murugesan, 1999]. The Internet and the Web provide software developers an easy access to real time data and alleviate access to software development tools. Internet technologies allow distributed networking, global access, platform independence, information sharing, and internationalisation [Gao *et al.*, 1999]. Development teams are now able to work from all over the world on same software project 24 hours per day. This increases the possibility of using personnel and competence in more efficient, flexible, and comfortable manner [Asklund, 1999].

3.2. Main concepts

A g*lobal software team* is separated by a national boundary while actively collaborating on a common software or system project. Two or more global software teams are implementing software in global software development projects. Each of the locations where these teams are working is called a *site*. Usually every site has one or more servers that are connected to the servers in another sites. The team members can be connected to the servers on their own site from a number of workplaces.

When developing the same software, all teams need to be able to access all needed source code, documents, and so on. All this information is stored in a repository. *Replication* is a technique to copy the repository from one server to another. Both the original repository and the copy are called *replicas*. The replicas are synchronised regularly to make the published changes visible at all sites. Some of the architectures that implement global software development use replication to share data between different sites.

The term global software development has replaced the previously used terms *geographically distributed development*, *distributed software development* and *software engineering over the Internet* mainly during the year 2001. The trend is not only shown in articles published of this area, but also in the name of the annual international workshop. The workshop was arranged four times with the name of “Workshop on Software Engineering over the Internet”, but in 2002 the name was changed to “International Workshop on Global Software Development”. The intention of the change was to broaden scope and address any issues of software development in global enterprises [Damian, 2002].

3.3. Global software development categories

Since companies are under pressure to increase productivity and reduce development time, they need to find new ways to save costs and time. Global software development has been seen as a good way to achieve these advantages. Saving time and costs are general reasons for global software development [McLaughlin, 1996]. It is possible to a global software development team to reduce development cycle time and lower cost, and also to improve quality and foster innovation [Carmel, 1999]. Often it is not an intentional decision to start doing global software development. It is rather an outcome of some change in organisation. In many cases, it is due to a merge of two or more companies, an acquisition or a lack of competent employees. The reasons that lead to global software development are divided into four categories in this thesis: multinational organisation, subcontracting, partnerships, and employment issues. Figure 5 represent these four categories and the reasons belonging to each of the categories. The categories and the reasons are explained more in the next paragraphs.

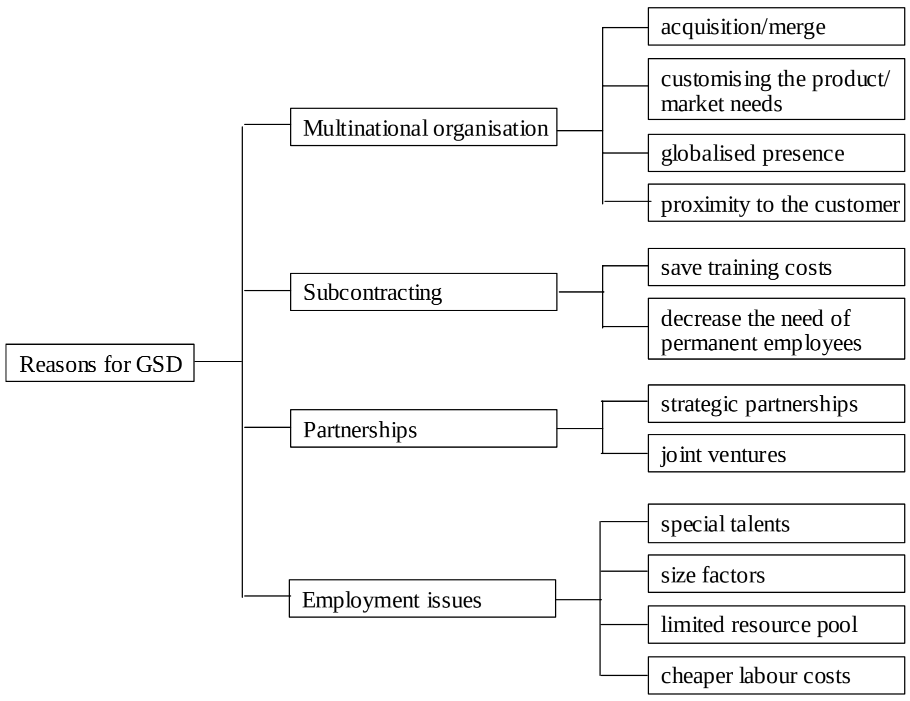

Figure 5 The reasons for global software development.

Multinational organisation is an organisation that has development groups in two or more countries. This can be due to an acquisition or a merge of two or more companies. The growing number of acquisitions and merges adds new markets, products, engineers, and creativity to the existing team. It can mean that software teams at other sites of the globe are suddenly forced to collaborate. A multinational organisation has no need or possibility to relocate all people to one geographical location. Thereby, large projects can be split among the teams at different sites. Another cause to form a multinational organisation is to set up an office to another country for some special reason. The reason could be customisation of a product or a special need on the markets in that area, for example. Other common reasons are proximity to the customer and globalised presence. Proximity to the customer is important, since software requires lot of interaction between designer and customer. This is ideally handled in face-to-face meetings. However, usually it is enough if the company is this close only to the main customers. Globalised presence is important because it is a strategic signal from the company to the world that they are a global player [Carmel, 1999].

Subcontracting is a fact of life for many organisations [McLaughlin, 1996]. It is common to use a third party that has experience in a specialised area of software engineering or some other form of excellence. By outsourcing development activities, an organisation can save training costs. Thereby they do not have to invest in knowledge, which is outside the core competence of the company. In addition to software development, subcontractors can be used to other tasks like secretarial work and technical writing. Thus, subcontracting decreases the need of permanent employees.

Partnerships are the third category of reasons for global software development. There are two kinds of partnerships: strategic partnerships and joint ventures. Companies use strategic partnerships to develop and promote their software products to gain better market access and to avoid becoming too large. Too large software development centres are unwieldy to manage. Joint ventures are formed to bring together different expertise of technology or capital and resources [Karolak, 1998]. One partner may bring equipment capital, while the other provides technical resources. One partner may bring expertise in one type of technology, while the other brings in another technology. In joint ventures, the partners start a separate company. The company tends to have more financial pressure and thus aims to develop software at a lower cost.

Employment issues can lead to global software development. Finding and hiring specialised talents is one reason. A software company may need to hire the best software developers regardless of their geographic location. Size factors are another thing. When one development centre is becoming too large, a solution can be to expand in another city or country. For large software companies it is hard to find competent people from a single R&D centre, for example. The resource pool is limited and there are several other organisations competing in the same labour market. Thus, the company needs to expand to a new location. Many companies are taking advantage of the cheaper labour costs of third world countries [McLaughlin, 1996]. One popular new R&D centre is India, which has cheap labour costs and a lot of technical universities educating competent and fluently English-speaking engineers.

3.4. Challenges

Global software development has advantages as described earlier. However, there are also several challenges to consider. Haag *et al.* [1997] think that geographically distributed software processes have increased the magnitude of the problems to be addressed. A part of the problems is due to the fact that the process is induced by the changes in the social and economical environment of software engineering. A good example of the increased magnitude concerns defect detection. Studies show that teams, which sit at the same place, need only a half the time for the defect detection compared to geographically dispersed teams [Ebert *et al.*, 2001] [Herbsleb *et al.*, 2001].

There have been many projects studying the challenges of global software development. Some of them target on technological aspects. Others target more on non-technical issues, such as communication and coordination. Dale W. Karolak [1998] has divided these challenges in three main categories in his book: *organisational*, *communication*, and *technical*. After analysing several other studies [Carmel, 1999] [Hofstede, 1997] [White, 2000] I decided to add a category for *cultural* challenges in addition to these three in this thesis. The following paragraphs present these categories as the four categories of global software development challenges. Organisations can try to avoid these challenges by limiting the dependencies between sites. This is not always possible and usually some dependencies remain. I discuss more about how to overcome these challenges, especially in software configuration management point of view, in Chapter four.

3.4.1. Organisational issues

Organisational problems are about the roles and responsibilities of the project participants. They concern both different sites inside one company and sites working with subcontractor. An organisation cannot function without coordination and control. Coordination is the act of integrating each task with each organisational unit [Carmel and Agarwal, 2001]. Control is the process of adhering to goals, policies, standards, or quality levels. Unfortunately, geographical distance creates difficulties in both. Because of distance, people cannot coordinate by peeking into the colleague’s office. Also managers cannot control by walking down the hall and visiting subordinates. Coordination mechanisms (like architectures and processes) are one key role in overcoming distance in global software development [Herbsleb and Grinter, 1999].

According to White [2000] the key issues of coordination are:

- who is responsible of the whole project,
- who is responsible for the managerial issues,
- who is responsible of the overall system architecture, and
- who are the team members involved.

He recommends projects to establish a “super project” organisational structure. In a “super project”, many smaller projects are collaborating on an overall project. This is a good approach, since it is often difficult to integrate separated independent teams into a coherent team. In addition, it may be that people outside the traditional product organisation staff the project [Battin *et al.*, 2001]. This creates a significant risk of the lack of problem domain. For the manager of the “super project” there is a difficult decision to make: how to divide up the work across sites.

Often organisations resist global software development. People may believe their jobs are threatened. They experience a loss of control and fear the possibility of relocation. Battin *et al.* [2001] reported in their survey that the management team had a genuine concern that the international engineering teams would not be able to produce as needed.

McLaughlin [1996] emphasizes that someone should be assigned to monitor the development of the whole project. Otherwise there can appear seemingly unrelated problems that are reported separately at each site without anyone realising the real issues. He gives examples of situations where these problems easily occur. One situation is when there is rarely anyone monitoring the project progress on a technical level. Another situation relates to subcontracting. With subcontractors there often appear additional problems when the build of the whole software is started.

3.4.2. Communication issues

Communication problems also hinder global software development. These problems involve the technical infrastructure the team members use to communicate with each other [Karolak, 1998]. Distance makes coordination and control problems worse through its negative effects on communication [Carmel and Agarwal, 2001]. However, several studies show that communication is the key to a successful project and that inadequate communication creates most challenges [Battin *et al.*, 2001] [Damian and Zowghi, 2002] [Haywood, 2000]. Haywood’s [2000] surveys confirm that success is more likely when people emphasize improving their communication as much as improving the tools they use.

In global software projects, communication is weakened, because the teams are geographically separated. People in the teams may speak different native language, they have experience on different processes, and they have different training backgrounds. The geographic separation makes it impractical to ever get the entire team together. Thus, it takes time to get information from all members of the project and the interactive conversation within the project is limited. The communication problem is not only with team members in different cities or countries. In fact, being in another building or even at the other end of a long corridor severely reduces communication [Herbsleb and Moitra, 2001]. Tom Allen [1977] found in his study that communication drops precipitously when offices of engineers are more than 25 meters from another.

Perhaps the most distinguishing feature of global software development teams compared to local teams is the need to find new ways to augment or replace the traditional face-to-face meetings [Ramesh and Dennis, 2002]. Today there are many new communication channels like e-mail, groupware, and video conferencing, but they cannot compensate face-to-face meetings in all cases. Face-to-face meetings are the richest communication media since it is a real time two-way interaction involving also nonverbal and implicit communication. It has been estimated that about 80% of the message that we communicate is other than explicit text. The bigger part consists of body language such as gestures, facial expressions, and posture. A recent study of dispersed software teams found that team members always wanted a richer communication medium no matter what the task [Carmel, 1999]. However, other media have their advantages. For example, e-mail provides the ability to explain details and keep written record and history of issues. But the downside of e-mail compared to traditional meetings is its lowered ability to handle ambiguity and that e-mail can be forgotten [Damian and Zowghi, 2002].

As mentioned above, traditional face-to-face meetings play a big role in communication of local software projects. But a surprisingly large role is also on informal face-to-face communication. Asklund [1999] has reported that an astonishingly large part of information is covered during discussions at review meetings, during coffee breaks, in the corridors and so on. Informal “corridor talk” helps people stay aware of what is going on around them [Herbsleb and Moitra, 2001]. It provides people with many essential pieces of background information that enables working efficiently. When those communication channels are missing there is a risk that connectivity within the group will be weakened. Understanding people’s motives, agendas, and other human interactions will be more difficult. Damian and Zowghi [2002] report from their field study that without the informal talks it is harder for project members to become a team. Since “you need to know each other personally to trust each other, to see a value of a person, to become engaged and committed”. Distributed team members may mistrust each other due to excessive stereotyping and lowered interpersonal attractiveness [Carmel, 1999]. One solution to ensure effective communication is to keep global teams small. Small teams also associate with the sense of intimacy that creates trust and cohesiveness.

The global software development team members should have a common understanding on how to prioritise the communication media they use [Haywood, 2000]. Prioritising communications builds trust among team members. Any organisation that is developing software in a geographically distributed way should also pay close attention to the costs, methods, and procedures associated with communication [McLaughlin, 1996]. A global way of doing business requires an investment in people and training that emphasises effective communication.

3.4.3. Cultural issues

Different cultures and different development styles introduce additional obstacles in global software development [White, 2000]. Managing cultural differences can only be achieved by awareness of the fundamentals of cultural differences [Carmel, 1999]. However, stereotyping about cultures and work styles can lead to misinterpretation of actions. But the cultural differences do not confine only to the differences between nationalities. There are also other types of cultures such as ethnic, corporate, and professional cultures. Each individual is a member of multiple cultures and each of these cultures has a different kind of grip on them. The corporate culture and the professional culture can make things easier in global software development, when all members of the project are from the same company and have the same profession. But it is still essential to be aware of the differences of the national cultures.

Cultures differ in many dimensions, such as the need for structure, attitudes towards hierarchy, sense of time, and communication styles [Hofstede, 1997]. In some cultures, employees are careful about expressing their opinions to superiors and show proper respect to the boss. In individualist cultures, people are concerned with personal achievements and independence, while in collectivist cultures people see themselves primarily as a part of the group. The cultural attitudes towards business versus softer values are most apparent when comparing Japanese and Scandinavians [Carmel, 1999]. The Japanese norms value long work days and little utilised vacation time. Instead, Scandinavians have a 38-hour workweek and long annual vacations. Some cultures avoid high risk and place greater emphasis on stability rather than innovation and change.

3.4.4. Technical issues

Technical issues are a fourth group of challenging issues of global software development. These issues involve the methods and tools used to solve technical problems. First, there is a risk to become dependent on slow and unreliable network [Asklund, 1999]. Tasks that involve transmission of critical data and multisite production must be planned and executed precisely. Software configuration management is one example of such tasks.

Another risk is differing technologies. It may be possible for all teams in one global project to use different technologies. However, it can have significant effects on usability, ease of installation, and look and feel, for example [White, 2000]. Also problems in system interoperability and usage of incompatible data formats can be complex and time-consuming. Thus, it is important to decide at some level what kind of technologies and standards teams are going to use. It helps planning and discussion when common concepts and terms are agreed on. Commonality in these areas provides a common practise that unites developers across language and cultural barriers [Karolak, 1998]. The development tools should also have the same version, if all teams decide to use same tools. Having same version at all sites can be difficult, since the latest version is not always available at the same time in all countries. In addition, it is important to assure that all sites can obtain support from the tool vendor. Simultaneous tool updates at every location may be costly and cumbersome to coordinate and control [Murugesan, 1999]. When development tools for global software projects are being chosen it must be ensured that they are able to support global software development [White, 2000].

Security is also a technological issue. The information of organisation is company proprietary. Thus, in global software production most information repositories should only be accessible from within the organisation intranet [Gao *et al.*, 1999]. A rigorous security mechanism must control the access of users outside, if needed.

3.5. Models of distributed software development

There are different ways to implement global software development. Different projects can use different ways depending on the size of the project and the technical possibilities available. The ways are divided into four models in this thesis according to Asklund [1999]:

- *distance working*,
- *subcontracting*,
- *co-located groups*, and

- *distributed groups*.

Each of these models can occur alone in a project or there can be combinations. Despite of the name, the third model is not necessarily used in all subcontracting cases. Subcontractors can also use distance working or co-located groups models.

Distance working means usually that some short task is done somewhere else than in the office. It can be done at home, for example. For this model it is typical that only slow network connection is available, but the person has a need to establish the working environment fast. There are two ways to do distance working: either a person can bring the needed files home as a copy or she can take a remote connection to the office.

Subcontracting means that a third party is bought to develop certain parts of the software. Subcontracting is based on close co-operation between the customer company and the subcontractor. Usually the customer company provides an environment to the subcontractor where the subcontractor can test the components before delivery. The customer company is responsible of the whole product and it controls errors and changes also to the parts developed by the subcontractor.

The notion of *co-located groups* means that developers belong to a local group or project. The work is divided between groups so that every group is as independent as possible. Division enables working locally as long as possible without communicating with other groups. Groups usually have access only to the latest stable version of other groups' outputs. In global software development, updates and deliveries between groups demand more consideration and administration. It can be thought of as an inner delivery, which tends to come infrequently. Co-operation between groups is easier if work is divided to phases that all developers are aware of.

Distributed groups mean that not only groups of developers work in different sites, but also developers in one group can be located at different sites. There is no daily communication even inside one group. Decreased communication can cause problems when several developers inside one group want to modify common components at the same time.

3.6. Architectures

Usually software is developed locally on a server. All developers are in the same place using a fast network connection and the same server. In geographically distributed development, there is a need for different kind of architectures. Asklund [1999] introduces five types of architectures:

- *remote login*,

- *laptop computer to a server*,
- *several sites by master-slave connections*,
- *several sites with differing areas of responsibility*, and
- *several sites with equal servers*.

The architecture to choose in development phase depends on the tasks, on the organisation structure, and on the policies that the organisation follows. Architecture has significant influence on the developers' awareness of the product phases. Long time intervals between synchronisations prevent changes becoming effective and thus control the degree of awareness.

Remote login is typically used in a situation when all developers are connected to the same server, but some of them are located elsewhere than where the server physically exists. Figure 6 represents the remote login model. Dashed line describes the site where developers normally work. All workstations are connected to the same server where different subcomponents are stored. Small circles represent subcomponents. Developers located elsewhere than the server can use telnet, for example, to get connection to the server. Technically this is a same situation than if all developers would be locally connected to the server. Only the network connections are slower.

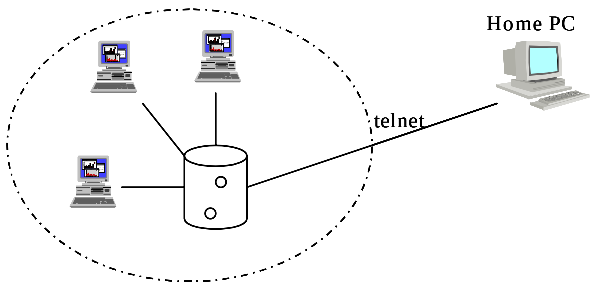

Figure 6. Remote login.

Laptop computer to a server describes a situation where some files are copied from the server and are developed locally. Figure 7 represents this model. Developers work normally at one site. There they are connected to the server where subcomponents are stored. A developer can take a copy of the needed files to be developed locally. She can take the copy to the laptop, for example. This case occurs when the developer takes some files with her to home or while travelling. Copied files are updated and synchronised every day or less frequently. The local development is usually done without the support from the software configuration management tool.

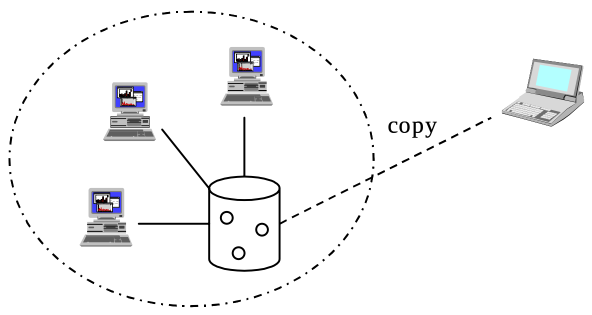

Figure 7. Laptop computer to a server.

The notion of *several sites by master-slave connections* means that one version from a subcomponent is copied from the master server to a slave server. The copy is developed further on the slave server. Figure 8 represents how the master server on the left has several subcomponents and one of them is copied to the slave server on the right. There can be several weeks or even months between updates. This architecture is typical in subcontracting, for example. It may be that there is no software configuration management tool or a different software configuration management tool in the slave server. Thus, version history does not get copied in the updates and the master site usually does not allow changes to the subcomponent.

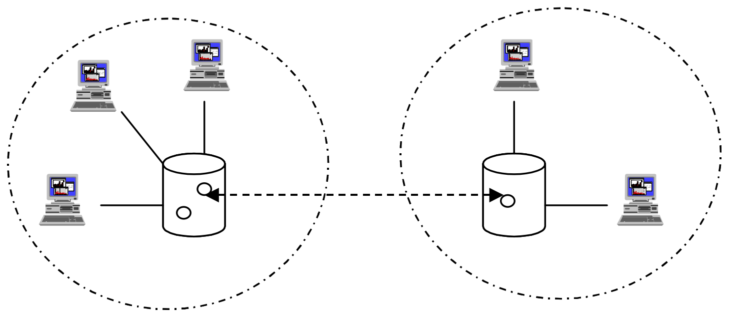

Figure 8. Several sites by master-slave connections.

Several sites in different areas of responsibility restrict access rights to components from those sites that do not develop them. Sites have no write access to files belonging to subcomponents they are not responsible of. In Figure 9, subcomponents are represented with a dashed line on that server where they have a read only status. Synchronisation is done by copying changes from the original to the replica. This can be done inside a software configuration management tool or manually with application level protocols like ftp or http. Usually replicas are automatically synchronised. Synchronisation takes place regularly or when needed. The same server can have both original subcomponents and replicas from other subcomponents. This means that updates can happen in both directions. Compared to master-slave connections the division between sites is more permanent and synchronisation is more automatic and frequent.

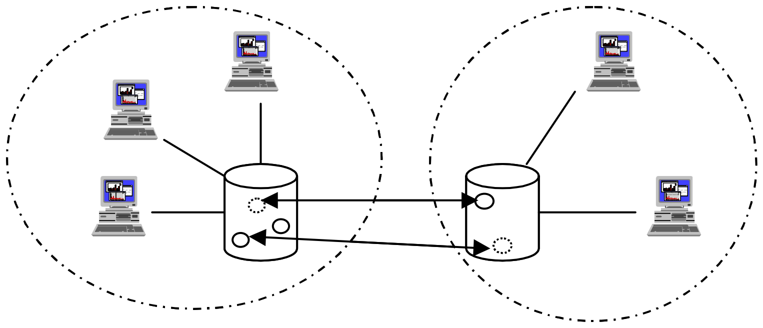

Figure 9. Several sites in different areas of responsibility.

Several sites with equal servers can be thought of as an ideal situation. It means that there are equal kinds of servers in several different sites. The servers are synchronised automatically between short time intervals and they all contain the same information. Figure 10 represents the situation where two sites have equal servers. There is an identical copy of both subcomponents on both servers. Developers can work at any site they like without noticing any difference.

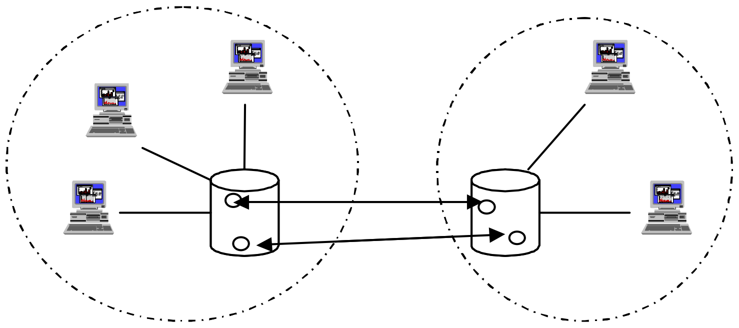

Figure 10. Several sites with equal servers.

3.7. Combinations of architectures and models

There are a couple of suitable possibilities in the architectures to choose from when implementing each of the models of global software development. In cases of co-located groups and distributed groups, there is one option that suits best, but distance work and subcontracting can be implemented with more than one architecture. All combinations are described below and illustrated in Figure 11. Shaded boxes in the figure show the suitable combinations.

	Remote login	Laptop computer to a server	Several sites by master-slave connections	Several sites with differing areas of responsibility	Several sites with equal servers
Distance working	🟦	🟦			
Subcontracting	🟦	🟦	🟦	🟦	
Co-located groups				🟦	🟦
Distributed groups	🟦				🟦

Figure 11. Different combinations of models and architectures. Shaded boxes illustrate the suitable combinations.

Distance working is usually implemented either with *remote login* or with *laptop computer to a server*. Other architectures are not suitable, since this model usually concerns only one employee at a time. *Remote login* is a good option, when distance working means working from home. The employee can have a network connection (like ISDN or ADSL) from home to work. She can use the real development environment and necessary tools from the same server where the work is normally done. A *laptop computer to server* is a better choice if the employee is working elsewhere than home, like when travelling. Working with the necessary files is faster, but the real development environment is not available.

Subcontracting can be implemented with all other architectures except *several sites with equal servers*. Equal servers would mean that the subcontractor has an access to all source files of the customer company. That is usually not wanted nor it is necessary. Subcontractors can use a *remote login* to customer company's server if the replication of sources is not possible for a lack of support from the software configuration management tool, for example. When there is only one person from the subcontractor company doing some small task to customer company, she can use the *laptop computer to server* architecture. The most typical architecture for subcontracting is *several sites by master-slave connections* [Asklund, 1999]. Customer company's server gives a copy of necessary source files to the subcontractor's server where the development can continue in isolation. The source files are copied back to server of the customer company when the development tasks are ready. *Several sites in different areas of responsibility* is a good option, if the subcontractor is working in a more close relationship with customer company. Customer company can restrict the access rights of subcontractor from those parts of the repository, which they do not want the subcontractor to be able to read and write. In this architecture, the replication is more frequent and source files are more up to date at both sites.

With co-located groups the most suitable architecture is *several sites in different areas of responsibility*. Since one group is always located at one site they do not need write access to source files developed at other sites. Each site has access rights to those files they need to modify and they can also read and use files from other sites for compiling, for example. *Several sites with equal servers* could also be used but the additional value it brings is not necessary. Other architectures are not appropriate for co-located groups. If every developer would use *remote login* or *laptop computer to server* architecture it would mean extra work and wasted time. With *several sites by master-slave connections* the replication happens infrequently, which can slow down the development needlessly.

Distributed groups model is most effective if the chosen architecture is *several sites with equal servers*. *Several sites with equal servers* is the only architecture that enables developers to modify their code from every site when ever they want. Since the communication between developers in one group is decreased it could be very risky if developers could not trust that the source files they see are always up to date. *Remote login* is another choice as the architecture in distributed groups model. Its drawback is slow working, if many developers need to login to a server on another site. In other architectures, the source files would be updated so rarely that they cannot compensate the reduced communication. In addition, developers could not work without the risk of overlapping and conflicting changes.

3.8. Summary

In the last decade of the 20th century, software industry had to face new challenges. There was pressure for better quality, increased productivity and reduced development time. At the same time Internet was developing rapidly. Organisations were growing bigger due to fusions and subcontracting became more popular. Because of the huge growth of information technology business, IT-companies needed much more employees. However, the traditional R&D centres had reached the limit of their labour pools. One option for organisations to find enough competent employees was to expand to other geographical locations. There are emerging centres of talented software engineers in countries like India, China, Philippines, Russia, and Hungary. These countries have also much cheaper labour costs, so they are a tempting choice for a company to expand. These are some of the reasons why global software development has emerged.

Companies try to achieve cost- and timesaving by using global software development. However, there are challenges like organisational, communication, cultural, and technical issues. Organisational issues relate to problems caused by unclear responsibilities and roles. Communication issues include decreased possibility to informal and face-to-face communication. Cultural problems are due to different cultures and they cause poor communication and misunderstandings. Technical challenges relate mostly to problems with network connections, development environment, and development tools.

There are four different models to distribute development geographically: distance working, subcontracting, co-located groups, and distributed groups. Organisation can choose to use only one of the models or to combine them. Remote login, laptop computer to a server, several sites by master-slave connections, several sites in different areas of responsibility, and several sites with equal servers are the five architectures to implement the models. Models and architectures can be combined according to the needs of the organisation.

Global software development is nowadays a fact in many organisations. Organisations have noticed that it is not an easy task to control geographically dispersed software projects. Keeping the repository up-to-date and all developers aware of the implemented changes is a difficult task, when the number of communication channels is reduced and the number of repositories is increased. Software configuration management is the area, which takes care of the repositories and has tools that can assist on information sharing.

4. SCM in global software development

As seen in Chapter 2, software configuration management is a discipline for controlling the whole evolution of software systems. It helps to deliver highly functional quality software in time and to budget, and helps with the development, support, and maintenance tasks in the longer term. A software configuration management system helps to improve the productivity and decreases the ramp up time of new employees. Today most SCM systems cover all the areas of software configuration management and help to maintain control over the evolution of the software project. However, many of the systems lack a proper support for global software development. The commercial software configuration management tools do not have complete solutions to overcome and deal with all the challenges global software development brings to software projects.

Chapter 3 described how global software development is a fact in many organisations nowadays. The possibility to save costs and reduce time-to-market has increased the amount of organisations involved in global software projects. Yet, global software development has disadvantages that prevent organisations to gain most of the benefits it brings. There are many organisational, communication, cultural and technical challenges, as described in section 3.4, which can cause global software development projects to be unsuccessful. Also the chosen geographical distribution architecture can have impact on how much and in what way those issues affect the software project.

The next sections concentrate on discussing what kind of an impact the four challenges of global software development have on software configuration management. It seems that some of the areas of software configuration management contain the same features that were discussed in the challenges section of chapter three. These features are the demand of formalism, capability to track changes, and control of the configuration item repositories, for example. After this analysis, we move on to explore what kind of an impact the eight different ways to combine geographically distributed development models and architectures have on software configuration management. Section 4.7 presents a new theory to overcome the barriers of global software development. I believe software configuration management systems have a significant role in this theory.

4.1. Organisational issues and SCM

The organisational issues relate to the roles and responsibilities of the members of the software project. In global software development, the major concentration in organisational issues should be focused on coordination and control, as discussed in section 3.4.1. Other concerns are the resistance of people towards sharing knowledge among global team members and monitoring the development of the whole project. Software configuration management is an essential tool for controlling and monitoring, since it has configuration control and status accounting activities.

For coordinating responsibilities White [2000] suggests that projects would establish a “super project” organisational structure and share the main responsibilities with the help of it. Ebert and De Neve [2001] add that one project leader should be fully responsible for achieving the targets of the whole project. The leader should have a project management team that represents the major cultures within the project. Software configuration management system is a tool for the project management team to control the project.

Monitoring the project on technical level should be planned carefully. Monitoring of the whole project can be easily neglected when multiple sites are involved with teams having different kind of responsibilities. There can be, for example, teams following up the performance of the network, teams taking care of the functionality of the SCM tool, and sub-project managers who follow up the progress of each sub-project. Each of these teams can notice problems at their site, which may seem unrelated. However, all these problems may be caused by a common technical problem.

It seems that the main question in handling the organisational issues on global software development projects is how to follow the progress of the whole project. In addition, there is a question on how to clear up everybody’s responsibilities to all members of a project in a multisite organisation. These questions have clearly an impact on software configuration management. Software configuration management is in a key role when project members are solving these questions, because it holds the information of what has happened and by whom. SCM systems collect and capture the history of changes made to the software and they contain the information of who was responsible for which change. These features are even more important in global software projects than in local projects, since changes are made on multiple sites. Therefore it is obvious that project members are going to use these status accounting features more in global software projects. SCM systems have to allow this information to be accessible from all sites that need it. Managers must be able to follow the progress of the project and the status of the work on each site. Following the progress and the status must be easy, and it must be possible to do these tasks from the site the managers are located at. When software configuration management is used in process control to help coordination, it is important that the functionality of the SCM tool is configurable by a user of different projects and project’s phase as well as user and user groups, roles, sites and responsibilities [Cocchio and Puttero, 1999].

Haag *et al.* [1997] report in their study that 30% of organisations valued software configuration management highly as a suitable solution for coordination in large projects. I believe that there are still possibilities for SCM systems to enhance in this area. Many of the commercial tools are still too complicated to use for managers, for example, because they use the tool so infrequently. The information is not easily available, but the person needs to execute several commands to find what she is looking for. I believe that the tools would be utilised more, if they would offer an easy to use Web access to the repository, for example.

4.2. Communication issues and SCM

When an organisation is distributing software development in geographically different locations, the communication between developers usually decreases and becomes more formal. The interpersonal relations are more difficult to maintain in large environments and formal procedures are required since humans cannot cope with the increased volume of information [Grudin, 1988]. Still the need to exchange information is much higher in global software development environments compared to local development [Cocchio and Puttero, 1999]. There are many technical solutions available like e-mail and video conferencing to help communication. However, they require extra work and they do not compensate the informal communication channels. On the contrary, e-mails can easily cause information overflow by providing too much information.

Haag *et al.* [1997] noticed that software engineering community is placing most emphasis on technological issues faced by global software development teams and less attention is paid to informational problems. The main goal in activities supporting communication is that people get all the information they need at the right time. At the same time they should be protected from any information they do not really need. It should be ensured that all material produced during the project is properly controlled and that only the necessary material is available at each site [Cocchio and Puttero, 1999].

In global software development, the communicational issues can be encapsulated to two main problems: decreased informal communication and increased amount of formal information. Software configuration management systems try to support collaboration by providing information about what other developers are doing, structuring development work, and automating various development activities [Grinter, 1996]. Thus, both of the problems above bring challenges to software configuration management. Project members need to use more the configuration control activities of software configuration management to compensate the decreased informal communication and to be aware of what has happened in the development work done by others. A software configuration management system can be an easy tool for developers to find the information they need from the information overflow. On the other hand, the decreased informal communication can cause misunderstandings. And misunderstandings lead to overlapping or unintended changes, for example. At the same time the amount of formal communication required can feel annoying to the project members and decrease the motivation to fulfil the necessary configuration identification and control activities of software configuration management. In global software development, the SCM system should pay attention to these kinds of situations, especially when changes affect configuration items in multiple sites. The possibilities to integrate e-mail or other communication tools to software configuration management systems should also be considered. It could help reducing the information overflow by sending the information only to the people concerned.

4.3. Cultural issues and SCM

Cultural issues in global software projects concern mainly different national cultures and different development styles. But in cases of subcontracting, partnership or acquisition, for example, it can also involve different corporation cultures. However, the software professionals worldwide belong to the same professional culture. Studies show that this computer subculture is stronger than national culture and in that way it reduces the amount of cultural problems [Carmel, 1999].

The different national cultures can have impacts on software configuration management. The SCM system is not effective and useful, if all the project members are not dedicated to use it. All the project members need to understand why the SCM system is in use and why it is necessary to use it appropriately. Thus, in multicultural projects the training of the software configuration management system must be done carefully. The cultural differences must be taken into account and assure that all project members have same understanding of the importance of the SCM system.

4.4. Technical issues and SCM

Software projects that are developed in many geographically different places, which are far away from each other, are highly dependent on network connections. Today network bandwidth is high and costs have decreased so that all developers can have almost immediate access to the sources [MacKay, 1995]. Still network failures happen and this hinders developers' work. Network failures cause problems in software configuration management. Most SCM tools cannot reliability transfer information or recover if a network failure happens during information transfer [Cocchio and Puttero, 1999].

Corporate firewalls cause another technical problems. Firewalls protect companies from unauthorised connections from public Internet to company's intranet. A firewall consists of a dedicated machine with special security precautions on it. It protects a cluster of more loosely administrated machines hidden behind it. A typical firewall machine has modems and public network ports on it, but it has just one carefully watched connection back to the rest of the cluster. All the software configuration management tools do not support replication of the repositories through firewall directly. It will probably need some extra adjustments before the replication works satisfactorily.

In global software development, the software configuration management system must handle the fact that development is done at multiple sites. Multi-site projects need an SCM database that controls the material produced at each site. This database contains read-only copies of material produced at other sites, fault reports, change requests, and modification reports for material, which is either produced at the site or used at the site [Cocchio and Puttero, 1999]. Thus, SCM systems need to cope with network failures, security issues and firewalls. Thereby, it is a good idea to dedicate a person to concentrate on these technical aspects.

4.5. SCM tools and global software development

On technical level one requirement from an SCM tool in global software development environment is replication of the repository. Allen *et al.* [1995] have presented few techniques to share common parts of the repository between multiple sites. The simplest approach is to provide all users at all sites the access to a centralised shared repository across a wide-area network. However, this approach is vulnerable to network problems, has unacceptable effect on system performance when using low bandwidth access methods, and presents problems with scaling the system to a very high number of users. Another option is to cache information locally at each development site by making use of a caching remote file system. A caching file system has the same problems as a central repository, but it can reduce the file I/O load imposed by remote access to version data on the central server. A third option is to replicate the entire repository to each local site. This way the sites may change their replicas independently.

Allen *et al.* [1995] introduce two different types of replicas: *serially consistent* replicas and *weakly consistent* replicas. Serially consistent replicas are continuously synchronised and avoid the possibility of lost or conflicting changes. However, this imposes a significant penalty on the availability of data in each replica: either reading or writing of data at any replica requires that, at least, a majority of all replicas are accessible. This majority-consensus requirement also means that the serially consistent replication approach has even worse scaling characteristics than using a central repository. Weakly consistent replicas allow the contents of individual replicas to temporarily diverge, with no guarantee that a change made at one replica is immediately visible at the other replicas. The presumption is that eventually the replicas will be resynchronised.

Replication requires automation to define a start time for the replication and to schedule synchronisation between sites, for example. Besides the technology and automation there are several important issues to think about to make replication efficient [Cocchio and Puttero, 1999]. What to replicate and where depends on the rights and responsibilities of each site. When and how often the replication should happen depends on the development processes used by sites. Thus, the software configuration management database should be able to model and track these processes either using its own features or by links to other process support tools. How to replicate depends on the infrastructure of tools and communication used by sites. Thus, the software configuration management database should be capable of being managed by a variety of tools, communicating across a variety of networks, and using a number of different replication strategies. It should support several ways of data transmission like online, temporary offline, and all-the-time offline as well as client-to-server, server-to-client, and server-to-server. Compression of the material replicated from one site to another is also required to be able to reduce transmission time and costs.

Parallel development of software components is common even in smaller software projects where all developers are working on the same site. Global software development brings an additional dimension to this by enabling a case where two developers working at different sites try to modify the same software component at the same time. Software configuration management tool should be able to handle these cases. One common way to handle parallel development is to use a branch and merge strategy, which can be used also in global software development. But globally distributed projects may require not just merging of some files but also merging of whole project structures [Cocchio and Puttero, 1999]. Branch names can include site information to help tracking down of changes.

There can be several different projects on-going at the same time in an organisation with many sites. There are large projects with many sites involved and smaller projects, which are developed in one site or at few sites. An SCM tool should be customisable so that it can be used satisfyingly with every project. It should be possible to customise the environment by selecting location and structure of projects, frequency of synchronisation as well as rights to access distributed functions [Cocchio and Puttero, 1999]. A software configuration management tool must also ensure security and privacy in data communications. Furthermore, an SCM system must take into account compatibility with corporate firewalls, as well. To maintain the tool flexibility, and to avoid the loss of customisability, the system should enable plug-in of proprietary encryption tools.

The software configuration management tool should be able to integrate with project management, CASE, process management, and development tools [Cocchio and Puttero, 1999]. Integration with reporting tools is useful to improve visibility of the development process as recorded by the SCM tool. Integration with project management tools means interacting with these tools by transferring tasks into them. Integration could automate the scheduling of implementation and could collect productivity metrics, which is very useful for project managers by bringing transparency into the project.

4.6. The impact of different models and architectures on SCM

As illustrated in section 3.7, there are eight different ways to meaningfully combine geographically distributed development models and architectures:

- distance working and remote login,
- distance working and laptop computer to a server,
- subcontracting and remote login,
- subcontracting and laptop computer to a server,
- subcontracting and several sites by master-slave connections,
- subcontracting and several sites with differing areas of responsibility,
- co-located groups and several sites with differing areas of responsibility, and
- distributed groups and several sites with equal servers.

In the next paragraphs, I discuss how these cases differ from software configuration management point of view. I present the demands they bring to software configuration management systems compared to local development. I also make some suggestions on how to resolve these issues.

Distance working can be done locally using a *laptop computer to a server* or remotely using *remote login*. When working locally, a developer is doing her work outside of the control and support of a software configuration management system. This means that the developer may need to create additional branches to configuration items before she can copy the files to her computer. An SCM system should have tools to help the synchronisation when changes are merged back to the original repository. The locally working developer is not aware of what others are doing and that can cause overlapping changes. Also, testing is impossible because the developer cannot see the work of others. Working remotely can cause extra branches, since the developer is working behind a slower network connection. It is more convenient to isolate work with branches since slower network connections can otherwise obstruct the work. There is a tendency in remote working that work models are not followed correctly but the work models could be integrated in the software configuration management system so that the system could force the developer to use them.

Subcontractors can work using *remote login*, *laptop computer to a server*, *several sites by master-slave connections*, or *several sites with differing areas of responsibility*. These architectures can have different impacts on the SCM system when combined with the subcontractor model. Here is presented only the most challenging case in software configuration management point of view. It is the phase where the components the subcontractor has implemented are integrated into the product, which may have been developed further. This situation is more complex when the subcontractor is working with *remote login* or *laptop computer to a server*. Despite of the chosen architecture, a customer company should manage the updating of the development and test environments. Performing the update can be difficult, if the subcontractor and the customer company use different software configuration management tools. The SCM tool can support the task, if the tools are same at both sites. Thus, at least *several sites by master-slave connections* and *several sites with differing areas of responsibility* architectures are easier to update.

The development groups are using the same software configuration management tool in case of co-located groups and *several sites with differing areas of responsibility*. Thus, this combination is an easier case. Nowadays also SCM tools usually support this architecture. Therefore, each group has a complete development environment and possibility to test their code. Files can still be on a different file system and groups more likely deliver sub-products between them rather than develop together. The most important issue in this case is that the communication works as well as possible between the groups. There are often few or no unplanned daily contacts between groups, but it is very important to all developers to have knowledge of the status between groups. In the software configuration management point of view, especially change management of common components is important.

Distributed groups and *several sites with equal servers* would be an ideal choice for global software development in developers' point of view. Unfortunately, there is no software configuration management tool that would support the architecture at the time. This architecture creates demands on how to implement the updating of the repository so that all configuration items are in the same state in every server without slowing down developers work with constant updates. In this case, communication is again important. All developers need to know what others are doing, how the project is proceeding, and which changes have been done and by whom. The SCM tool should have a strong support on division of files and on concurrent, simultaneous changes.

4.7. The object-oriented team model

An interesting approach on overcoming the barriers of global software development is presented in the article of Ramesh and Dennis [2002]. Ramesh and Dennis have analysed different global software development projects to find the key technologies and work processes of a successful global software development team. A traditional approach in research suggests using information rich media as much as possible to drive the project and to overcome the distance. Ramesh and Dennis found out a strikingly different pattern of interaction in their study. They term a team using this new pattern as "Object-oriented team" and the traditionally working team as "Integrated team". In the following paragraphs, I will present the Object-oriented team model, since it is very interesting in point of view of this thesis. The integrated team model is left out, since it is not in focus of this thesis.

The Object-oriented team strives to decouple team members through the use of semantically rich media. The decoupling is meant to decrease the ripple effects of changes in tightly coupled systems. The Object-oriented team model tries to avoid this tight coupling using a set of independent objects. These objects

- have a standardised or well-defined processes,

- exchange information with other objects through well-defined semantically rich interfaces, and
- produce a decreased flow of information.

The Object-oriented team uses well-defined processes and data. Each phase of the software project has clearly defined inputs and outputs, and rules for performing the work. The use of standard templates for documents is one key factor in facilitating effective communication among team members. However, while the processes and data are well defined, the assignments faced by the team are not.

The communication and coordination among the Object-oriented team members occurs mostly through well-defined messages passed via semantically rich media. Semantically rich media enable the transmission of information in containers that provide meaning beyond the information itself. These digital containers clarify, extend, and constrain the meaning of the information so that it is easier for recipient to understand. All the interviewees of the study of Ramesh and Dennis agreed that the availability of the semantically rich repositories was “the most important thing”. From a theoretical perspective, these semantically rich media enable the sender to edit and rehearse the information to ensure the meaning is conveyed exactly as intended. They also enable recipient to reprocess the message multiple times until the correct meaning has been extracted. The semantically rich media usually enable the recipient to manage information complexity by providing search capabilities and different views of the information. Many of the media automatically collect statistics and enable analyses of the information that would be practically impossible to collect from less semantically rich media.

The use of well-defined processes and semantically rich media, enable the decoupling of team members and a reduction in the flow of information. Once complexity goes beyond a certain level, it becomes impractical to communicate with information rich media. Thus, semantically rich media are needed to reduce the unneeded flow of information. Semantically rich media enable the “selective push” or the “selective pull” of coordination information. In the Object-oriented team, communication and coordination occurs mostly using semantically rich media. Nonetheless, information rich media is used to supplement the semantically rich media.

As an implication, Ramesh and Dennis state that they need additional research on the Object-oriented team model to seek additional evidence about its applicability. One challenge for the future lies in understanding when each form of team is appropriate. Ramesh and Dennis speculate that the Object-oriented team model may be most appropriate for large and complex projects, while the Integrated team model may work on smaller and less complex projects.

In the point of view of this thesis, the Object-oriented team model is interesting. Software configuration management systems correspond exactly to the description of a semantically rich medium. Software configuration management systems have document, code and bug repositories, which track changes, include comments about changes and history data. They provide search capabilities and different views of the information, and collect statistics. As the semantically rich media play a major role in the Object-oriented team model, we may conclude that the software configuration management system can be in a key role of a successful global software development project. Nonetheless, the research work about the Object-oriented model is still in progress and the results of the future studies should be followed closely. There can emerge new demands and valuable observations to consider when designing the software configuration management systems for global software projects.

4.8. Implications

Today most organisations have software configuration management systems that cover all the areas of software configuration management. However, many of the systems lack a proper support for global software development. In addition, the commercial software configuration management tools do not have complete solutions to overcome and deal with all the challenges of global software development. Thus, the global software development has impacts on software configuration management and those impacts need to be analysed to produce a successful global software project.

As a conclusion of the sections above, I state that in successful global software projects the role of software configuration management is greater than in local projects. Communicating in regular group meetings and having informal conversations together with some simple software configuration management system, which is mainly used in version and change control, may manage local projects. Global software projects need a properly defined software configuration management system with a full-fledged software configuration management tool. The SCM tool needs to be easy to use, so that every project member at every site would use it. The tool needs to ensure secure data transmission over the public network. And the software configuration management system has to guarantee reliable processes that are followed correctly at each site. Otherwise, the SCM system cannot fulfil the challenges of global software development. The requirements global software development brings to software configuration management are collected from the sections above and divided in three groups: security, reliability and ease of use.

The requirements of security, reliability and ease of use relate to configuration identification, configuration control, and status accounting areas of software configuration management system. These areas have functions that need extra consideration when developing SCM system for global software projects. Figure 12 represents how the single requirements in these three groups divide between the SCM tool and SCM processes. The single requirements are described more detailed in sections 5.4.1, 5.4.2, and 5.4.3. In an ideal situation, a commercial SCM tool has solutions to all these requirements. However, usually some of the requirements need to be solved with rules and defined processes. In Figure 12, those requirements are listed at both sides. The chosen distribution model and architecture change slightly the influence of the requirements to these processes, but here the implications are covered on a more common level.

	SCM tool	**SCM processes**
Security requirements	access control network problems secure and private data transmission compatibility with firewalls	access control
Reliability requirements	dependencies synchronisation control change control system performance replication	dependencies synchronisation control change control
Ease of use requirements	easy change tracking integration to other tools customisable user-friendly user interface	

Figure 12 The requirements of global software development on software configuration management systems.

On security issues, the main concern is the access control. The software configuration management tool should handle properly the other issues, which include the possible network problems, the security and privacy issues in data transmission, and the compatibility issues with corporate firewalls. In global software development, the access rights must be given only to necessary people at all sites. It must be assured that these people really are able to access the configuration items regardless of where they are. Possible solutions are to grant the access rights in a centralised way, or from all development sites independently.

The requirement of reliability means that the members of the software project must be able to trust the software configuration management system. They need to know that all configuration items are safely stored, easily accessible, and nothing unexpected can happen to their work. In global software development, the difficult parts are dependencies between configuration items, synchronisation control, change control, system performance, and replication of the repositories. The dependencies between configuration items are inevitable, but in global software development the amount of dependencies between different sites should be as small as possible. Otherwise, the dependencies can cause complexity and slowness to the work. The synchronisation control deals with the parallel changes, but in global software development it has to cope with parallel changes happening simultaneously at different sites. If the software configuration management tool does not handle these situations automatically, the software configuration management system has to have procedures to avoid overwriting the other developer's work. The change control procedures have to solve issues like how to analyse the impact of the change at all sites and inform the necessary people. The Change Control Board needs to have representatives from all sites to be sure that the impact of the change is fully analysed. The system performance of the software configuration management tool should be optimal also when working from another sites. The repositories must be replicated frequently to guarantee that all necessary information is accessible and up to date at all sites. Members of the project should be aware of the replication frequency.

The software configuration management system should enable an easy way to see what has happened, who is the responsible person, and how she did it. This information is what all the members of the software project need. In global software projects, this information is even more important for the members to follow the status of the project and the configuration items at other sites. It may be that the members from different sites have not ever met each other, so it is difficult to know whom to contact. Thus, the information should be available through the SCM system. The software configuration management tool should provide a user interface, where it is easy to track changes from different sites. In global software development, it is important for the SCM tool to be customisable to different kinds of projects, since the project can be anything from a small project developed at one site to a large multisite project. The capability to integrate the SCM tool to other tools brings transparency into the project, which is particularly useful in global software projects. A user-friendly user interface will make it more tempting to use the software configuration management tool and utilise all the information the system gathers.

The software configuration management system should compensate the decreased amount of informal communication by forcing to increase the amount of formal communication. Software configuration management system should define processes that force members of the project to add additional information to the repositories to clarify the actual configuration items. This information may include descriptions of the items and their status, for example. The SCM system should also gather automatically as much information as possible to provide meaning beyond the information itself to represent the semantically rich medium. This makes it easier for the recipient to understand the information, but it should not become a burden to the sender. The information should also help managers to be aware of the status of the project at all sites.

5. Case study: RC2 project in Nokia Networks

In this chapter, I describe and analyse the methods used in a real-life global software development project. The project has been ongoing in Nokia Networks. The target of a project is to produce an optional module of a large software product. The project (and the product the project is producing) is called *RC2*.

Nokia Networks provides mobile, broadband and IP networks and related services. The company develops mobile data applications and solutions for operators and Internet service providers. One of the offered products is network management system. The network management system is divided in several independent hierarchical components and customers can collect the components they need to manage their network. RC2 project implements one of the upper level components.

RC2 it is being developed at three different sites in Finland. The project uses a common platform as a ground for the development. A partner company abroad maintains one part of the platform and the other part of the platform is developed at two sites inside Nokia Networks. The other site is in Finland and the other is in Central Europe. The development is mainly done in UNIX environment. Some parts of the project are done in Windows environment, including the part of the platform that is developed in Finland, but those parts are left out from the scope of this case study. RC2 is using Rational's ClearCase as the software configuration management tool.

5.1. Basic concepts of ClearCase

ClearCase is a software configuration management tool made by Rational. It helps software development teams to track the files and directories used to create software, and enables them to manage the development and build processes. It also enables them to re-create the source base from which a software system was built, allowing it to be rebuilt, debugged, and updated. ClearCase is specifically designed to support parallel development, whether it means isolating the work of one developer from others on a small team, developing multiple releases in parallel using different teams, or sharing a source code base between multiple teams at geographically distributed sites. [ClearCase, 1999]

Files and directories are called *elements* in ClearCase and they are stored in a repository called *versioned object base* (*VOB*). The historical versions of the files in the VOB are stored in data container files. The VOB database records the evolution of the version-controlled file-system objects, and stores the associated metadata. Elements can be accessed and changed using a *view*. A VOB contains all versions of a particular set of elements. A view selects a specific version of each element using a set of rules called a *configuration specification* (*config spec*). A VOB looks like an ordinary file-system directory tree when accessed through a view. ClearCase uses a checkout-edit-checkin model to manage changes to elements.

ClearCase MultiSite extends ClearCase by supporting parallel software development and software reuse across geographically distributed project teams. ClearCase MultiSite enables developers at different locations to use the same VOB. Each site has its own replica of that VOB. The set of replicas for a particular VOB is called a *VOB family*. Each replica includes a full set of data containers and a complete copy of the VOB database. At its site, a replica appears to be a regular VOB. Regular ClearCase use models apply to the use of replicas. At any time, a site can propagate the changes made in its own VOB replica to the other members of the VOB family, using either an automatic or manual synchronization process. Thus, the replicas in ClearCase MultiSite are weakly consistent. MultiSite can also be used at a single geographical location, to allow independent groups to work with the same development data, to enable interoperation in a mixed UNIX/Windows networks, or to be a backup mechanism.

5.2. Basic concepts of the SCM system

RC2 project is using a software configuration management system that is based on ClearCase, ClearCase MultiSite, and some custom scripts on top of them. All configuration items are created and stored in source VOBs. There are also additional VOBs, which contain tools for building the software product and tools for administrating the software configuration management system. Each configuration item has a main branch, which represents the principal line of development. Sub-branches can be created for special purposes. Common cases for sub-branches are parallel development, maintenance work and trials. In ClearCase, one site always owns each branch.

All members of the project need a view to access the configuration items in ClearCase. Usually every developer has an own personal view (or several views), but views can also be shared between developers. A view for building the software product is one example of a common view.

The change control system is based on the usage of labels. All versions of configuration items belonging to one change are marked with a change specific label. The change labels include the description of the change. Labels are also used to mark the new configuration of the software product after every build.

The numbering system in the label names of the software builds indicates the phase where the project is.

ClearCase records automatically a lot of information on what happens inside the tool. It stores information of who did a change and when it happened. Developers can compare different file versions, do searches, and look at the history of the configuration items. The members of the project can view the whole version tree of the configuration items and see the descriptions of each version, label, or branch type. ClearCase keeps also book on which versions were included in the build and how the derived objects were created.

The source VOBs of RC2 are replicated to the development sites, since RC2 is a global software project. The ownership of different branches in a version tree of a configuration item can reside in different VOB replicas. Thus, same source files can be modified simultaneously at different sites. The simultaneous modifications are done in different branches, since developers cannot modify a version in a branch owned by another site.

5.3. Used models and architectures

RC2 is divided into several smaller parts called subsystems. Each subsystem has one development team that is responsible of it. Most of the development teams are located at the same site in Finland. However, one subsystem is divided between a Nokia team at another site in Finland and a subcontractor team. The platform is developed in Central Europe and it contains the part the partner company abroad is producing. Thus, there are two combinations of models and architectures used in RC2: *subcontracting* with *several sites by master-slave connections* and *co-located groups* with *several sites in different areas of responsibility*. The subcontracting team and the partner company abroad are using the first combination when distributing work with the teams at Nokia Networks. The teams at Nokia Networks are using the second combinations when distributing work between them. Occasionally individual developers use *distance working* as their distributed development model. At that time their architecture can be *remote login* or *laptop computer to a server*.

Nokia Networks offers a development environment to the subcontractor team. The subcontractor team has an access to an environment maintained by Nokia, which contains a ClearCase server and subcontractor specific ClearCase clients [Nokia, 2001]. Nokia ClearCase administrator and the software configuration management responsible person in RC2 have set up the necessary software configuration management environment. They are also responsible of the updates and managing the possible errors in the environment. There are firewalls between subcontractors' hosts and Nokia Intranet. Hence, a subcontractor team never has an immediate access to Intranet and they can only see sources replicated to the server in the Extranet.

In the Nokia maintained environment, subcontractor teams can do the development either on their own hosts using their own SCM tool and procedures, or connecting to the ClearCase clients owned by Nokia [Nokia, 2001]. The subcontractor team in RC2 is working on clients of the Extranet. Thus, the whole version history of all the elements the subcontractor team has implemented is stored in ClearCase VOBs in the Extranet. The Extranet server is acting as a slave server and the Intranet server as a master server. Synchronisation is done several times per day.

For the platform part that is produced by the partner company abroad, the architecture is same but it is implemented slightly differently. The team in the partner company works at their own premises and use their own ClearCase servers and clients. The team members can decide themselves how to handle SCM inside their own environment, but they have to store the code in ClearCase VOBs. The ClearCase server at the partner company is acting as a slave server and the server at Nokia Networks is acting as a master server. The replication is done through firewalls at Nokia and at the partner company, so the firewalls need appropriate configuring.

The three sites inside Nokia Networks have divided the work so that they can be as independent as possible. The teams have access to the latest stable version of the subsystems the other teams are working on. They have no write access to the files belonging to the subsystems they are not responsible of. Replication between sites is done automatically and updates happen to both directions. The VOBs can have both original configuration items and replicas from other configuration items.

5.4. Impacts on the SCM system

The earlier versions of the network management system product were developed using RCS version control tool. The tool was enhanced with custom scripts to advance the change control procedures. Nokia Networks had several development sites already then and there was a growing trend to increase the cooperation between these sites. At some point, it was inevitable that RCS and the custom scripts were not enough to handle the demands towards the software configuration management system. The network management system product needed a full-fledged third generation software configuration management tool to respond to the new requirements. The tool needed to be more scalable and flexible, because the projects were growing and becoming more complex. It was also required that the tool would support global software development. The new tool that was selected was ClearCase.

One of the reasons to select ClearCase was that ClearCase offered an optional package of functionalities called ClearCase MultiSite. ClearCase MultiSite is specifically developed to support global software development projects. It has features that enable developers at different locations to access the same data containers and to distribute their development efforts. It contains features that print time stamps that reflect local time and features that enable a configuration item with multiple branches to be developed in different sites concurrently, for example.

As stated in chapter 4.8 there are three groups of requirements for software configuration management systems in global software development. The groups are security, reliability, and ease of use. Usually the software configuration management tool responds to some of the requirements. Rules and processes are defined depending on the capabilities of the used tool. Thus, an appropriate SCM tool is a necessity for a successful SCM system in global software projects.

5.4.1. Impacts of the security requirements

In the case of RC2 project, ClearCase provides the solution to the requirements concerning security. ClearCase handles the security and privacy issues in data transmission, it can recover from many network problems, and it is compatible with the firewall solutions. ClearCase has a security model and the way access rights are defined by the security policy devised by actual projects. This means that RC2 can define how the access rights are granted.

5.4.2. Impacts of the reliability requirements

ClearCase offers solutions to some of the reliability requirements. It takes care of synchronisation control, system performance, and replication after all sites have defined common procedures for these. But there were still few areas in reliability group, which needed consideration in RC2 project. The following problematic areas were found during the early stages of the RC2 project when the SCM system was taken into use.

The problems had to do with dependencies and change control. The dependencies between configuration items at different sites caused problems when compiling and building the software product. In the software configuration management system of RC2, the building procedures are based on the usage of the build avoidance. ClearCase supports build avoidance as an in-built feature. The idea is to reuse the existing compilation results when possible to reduce the overall building time. Thus, every time a developer compiles her published changes ClearCase saves the compilation results. When other developers later need the compilation results for their code, ClearCase looks for the saved compilation results and uses them instead of recompiling everything. This feature has reduced the building time of the whole product tremendously. However, sometimes build avoidance can be a burden to an individual developer because of the source code dependencies between sites. The dependencies can form a chain, where a code of one developer has dependencies on another configuration items controlled by other developers. These items may have dependencies on some more configuration items, and so on. This chain of dependencies can include multitude of configuration items from several developers. When a configuration item at the end of the chain is changed and compiled, the whole chain needs to be recompiled to have correct results. When some of these configuration items are at another site, the compilation time draws out very long. The reason for that is that ClearCase cannot look for the saved compilation results from the other sites. Thus, it needs to compile everything from the scratch and it takes a lot of time.

These compilation duration problems surprised the developers in RC2. When developer took the new label into use, it might happen that she was the first one to compile everything from scratch. Thus, it made the problem very unpredictable. The project had already minimised the amount of dependencies, but sometimes the problem still came up. There were two solutions in consideration. One solution was to save the compilation result to a separate VOB after every subsystem build. The developers would use these ready results in their compilations to avoid the long compilation times. The other solution was to create a script that tracks the need to compile the configuration items in the chain. The script would ask the developer what to do, if it notices a dependency to a changed configuration item at the other site and a need to recompile many configuration items in the chain. The possibilities would be to compile everything or use the older versions, which are compiled already. Thus, it would be the developer’s choice whether she needs the new results and is willing to wait or not.

The change control procedures lacked solution on how to inform the necessary people at all sites about the published changes. This communication problem is typical in the combination of *co-located groups* and *several sites with differing areas of responsibility*. In RC2, the approved and published changes are communicated using labels. Labels are included to the config specs, which then select the right version of the configuration items. Occasionally the information of the labels and what kinds of changes are included in the newest release was not communicated to all necessary people. However, when the information was communicated properly, there were additional problems. All of the changed files were not necessarily replicated to the other sites at time when the request to use new labels arrived. There were again two solutions considered. First was to define strict procedures for communicating the information about the labels, and the second was to create a script to do it. The procedures define who sends the information to whom and when. They also define a person whose responsibility is to check that the changes have been replicated to the other sites before sending the information. The script automatically sends an email to defined group of people when a new label is attached to the changed files. Similarly the script checks that the changes are replicated to the other sites before it sends the email.

The building of the replicated configuration items may fail due the differences in the development environments. In RC2, this problem was not so significant because of the chose distribution architecture. *Several sites with master-slave connections* makes it easier to synchronise the replicas, since both sites are using the same SCM tool to store their configuration items. A solution to this problem was to keep the development environments consistent. For example, storing the development tools into a VOB solves this issue. The tools in VOBs are labelled during the build, so that the labels show the correct version of them.

5.4.3. Impacts of the ease of use requirements

The requirements in the ease of use -group also bring out issues on the SCM system. Especially in UNIX environment ClearCase is not a very easy tool to use. ClearCase has plenty information on what has happened and how, but it is not an easy task to find relevant piece of information when the task is at hand. ClearCase has a command line interface and a graphical user interface to satisfy the needs of different kinds of users. Some people prefer to use shell commands. They are accustomed using them, they find that command line interface provides quicker way to do batch tasks, and they can do own aliases for the commands. Other people prefer graphical user interfaces. They like to visualise their work, and to select the commands from predefined menus. In ClearCase, the graphical user interface in UNIX is slightly slower to use than the command line user interface. The graphical user interface is good at some tasks like when looking at the version tree of a configuration item. However, it needs to be started separately, which is not very convenient. The major drawback is that custom scripts cannot be used via the graphical user interface.

In RC2, most developers use ClearCase with the command line user interface. Since the project has many custom scripts to guide the developer through the change control procedures, it is impossible to do the work using mainly the graphical user interface. This is a big limitation since different users of the project need different types of user interfaces. However, since the graphical user interface in ClearCase is very slow to use, it is not reasonable to try to configure it to show also the custom scripts. In this case, I believe that it is better to wait for Rational to improve the graphical user interface first.

6. Conclusions

During the recent decades the trend in software engineering has been towards global software development. A growing number of organisations have development sites in several countries, are using subcontractors, or forming virtual corporations. While the software projects are controlled with software configuration management systems, it is inevitable that the global software development has some impact on those systems. The intention of this thesis was to find what kind of an impact the global software development has.

I have used a constructive method in this thesis to clarify the theories and concepts of software configuration management and global software development. I have presented results from several studies of the impacts global software development has on software projects overall. From the results I have picked up the ones that relate to software configuration management systems and analysed them more. The conclusions of this analysis are presented here as results of this work.

The main impact global software development has on software configuration management is that in successful global software projects the role of software configuration management is greater than in local projects. Global software projects need a secure, reliable and ease of use software configuration management system. The most important thing is to select a software configuration management tool, which takes care of most of the requirements. However, each SCM tool needs also appropriate processes to fulfil all the requirements.

Depending on the SCM system, organisations can enhance the system by improving the software configuration management tool or by modifying the processes of the software configuration management system. If the SCM system is new, the organisation can satisfy most of the requirements by choosing a commercial software configuration management tool that best supports global software development and needs less defining of appropriate processes. Free tools usually do not have enough capabilities to support global software projects, but the third generation commercial tools already contain many necessary features. If the SCM system is already in use, it can be enhanced by defining strict processes or by building custom tools on top of the used software configuration management tool.

The requirements global software development has on security issues concentrate on access control, handling network problems, granting secure and private data transmission, and being compatible with corporate firewall. The software configuration management tool can handle all of these issues, but in access control also strict processes can fulfil the extra requirements. The requirements of reliability are handled by decreasing dependencies between configuration items, enhancing synchronisation control and change control to work in multisite environment, and optimising system performance and replication of repositories. The requirement on ease of use is even more important in global software development. Since the role of software configuration management is emphasized, the user interface of the SCM tool should be so easy to use that it would be tempting to the members of the project to utilise all the possibilities the SCM system has. It should provide easy ways to look and search information from all the sites involved in the project.

The software configuration management system can be the tool that compensates the decreased amount of informal communication. The theory of Object-oriented team is very interesting, since its goal is to decouple the team members and decrease the amount of necessary communication with the use of semantically rich medium. A software configuration management system fulfils the requirements of a semantically rich medium. Thus, it would be an opportunity for future studies to evaluate this theory and the possibility to utilise SCM system in it. This could be one solution to successful global software projects in the future.

References

[Allen, 1977] Tom Allen, *Managing the flow of technology*. MIT Press, 1977.

[Allen *et al.*, 1995] Larry Allen, Gary Fernandez, Kenneth Kane, David Leblang, Debra Minard, John Posner: ClearCase MultiSite: Supporting geographically-distributed software development. In: *Software Configuration Management: selected papers / ICSE SCM-4 and SCM-5 Workshops*, 194-214.

[Asklund, 1999] Ulf Asklund, *Configuration Management for Distributed Development - Practice and Needs*. Licentiate thesis, Dept. of Computer Science, Lund University, Sweden, 1999.

[Battin *et al.*, 2001] Robert D. Battin, Ron Crocker, Joe Kreidler, K. Subramanian, Leveraging resources in global software development. *IEEE Software*, **18**, 2 (March/April 2001), 70-77.

[Berlack, 1992] H. Ronald Berlack, *Software Configuration Management*. John Wiley & Sons, Inc., 1992.

[Bounds and Dart, 1993] Nadine M. Bounds, Susan A. Dart, *Configuration Management (CM) Plans: The Beginning to Your CM Solution*. Technical Report, Software Engineering Institute, Carnegie-Mellon University, 1993.

[Carmel, 1999] Erran Carmel, *Global Software Teams: Collaborating Across Borders And Time Zones*. Prentice-Hall PTR, 1999.

[Carmel and Agarwal, 2001] Erran Carmel, Ritu Agarwal, Tactical approaches for alleviating distance in global software development. *IEEE Software*, **18**, 2 (March/April 2001), 22-29.

[Chan and Hung, 1997] Angus K. F. Chan, Sheeung-Iun Hung, Software configuration management tools. In: *Proceedings of the Eight IEEE International Workshop on Software Technology and Engineering Practice incorporating Computer Aided Software Engineering*, 238-250.

[ClearCase, 1999] *Introduction to ClearCase*. Rational Software Corporation, 1999.

[Cocchio and Puttero, 1999] L. Cocchio, D. Puttero, Industrial requirements for distributed SCM tool. *Software Quality Journal*, **8**, 2 (October 1999), 111-120.

[Damian, 2002] Daniela Damian, Workshop on global software development. May 21, 2002. Available as http://www.cis.ohio-state.edu/~nsridhar/ICSE02/GSD/PDF/summary.pdf

[Damian and Zowghi, 2002] Daniela E. Damian, Didar Zowghi, The impact of stakeholders' geographical distribution on managing requirements in a multi-site organisation. In: *Proceedings of the 10th IEEE International Conference on Requirements Engineering*, 319-328.

[Dart, 1991] Susan Dart, Concepts in configuration management systems. In: *Proceedings of the 3rd International Workshop on Software Configuration Management*, 1-18.

[Do, 1999] Aimee G. Do, The impact of configuration management during the software product's lifecycle. In: *Proceedings of the 18th Digital Avionics Systems Conference*, 1.A.4-1 - 1.A.4-8.

[Ebert and De Neve, 2001] Christof Ebert, Philip De Neve, Surviving global software development. *IEEE Software*, **18**, 2 (March / April 2001), 62-69.

[Ebert *et al.*, 2001] Christof Ebert, Casimiro Hernandez Parro, Roland Suttels, Harald Kolarczyk, Improving validation activities in a global software development. In: *Proceeding of the 23rd International Conference on Software Engineering*, 545-554.

[Estublier, 2000] Jacky Estublier, Software configuration management: a roadmap. In: *Proceedings of the 22nd International Conference on the Future of Software Engineering 2000*, 279-289.

[Futrell *et al.*, 2002] Robert T. Futrell, Donald F. Shafer, Linda Isabell Shafer, *Quality Software Project Management*. Prentice Hall PTR, 2002.

[Gao *et al.*, 1999] Jerry Z. Gao, Cris Chen Yasufumi Toyoshima, David K. Leung, Engineering on the Internet for global software production. *Computer*, **32**, 5 (May 1999), 38-47.

[Grinter, 1996] Rebecca E. Grinter, Understanding the role of configuration management systems in software development. In: *Proceedings of the CHI '96 conference companion on Human factors in computing systems*, 39-40.

[Grudin, 1988] Jonathan Grudin, Why CSCW applications fail: problems in the design and evaluation of organizational interfaces. In: *Proceedings of the conference on Computer-supported cooperative work*, 85-93.

[Haag *et al.*, 1997] Z. Haag, R. Foley, J. Newman, Software process improvement in geographically distributed software engineering: an initial evaluation. In: *Proceedings of the 23rd EUROMICRO Conference: New Frontiers of Information Technology*, 134-141.

[Haywood, 2000] Martha Haywood, Working in virtual teams: a tale of two projects and many cities. *IT Professional*, **2**, 2 (March / April 2000), 58-60.

[Herbsleb and Moitra, 2001] James D. Herbsleb, Deependra Moitra, Global software development. *IEEE Software*, **18**, 2 (March / April 2001), 16-20.

[Herbsleb and Grinter, 1999] James D. Herbsleb, Rebecca E. Grinter, Architectures, coordination, and distance: Conway's law and beyond. *IEEE Software*, **16**, 5 (September / October 1999), 63-70.

[Herbsleb *et al.*, 2001] James D. Herbsleb, Audris Mockus, Thomas A. Finholt, Rebecca E. Grinter, An empirical study of global software development:

distance and speed. In: *Proceedings of the 23rd International Conference on Software Engineering*, 81-90.

[Hoek *et al.*, 1995] Andre van der Hoek, Dennis Heimbinger and Alexander L. Wolf, Does configuration management research have a future? In: *Software Configuration Management: selected papers / ICSE SCM-4 and SCM-5 Workshops*, 305-309.

[Hofstede, 1997] G. H. Hofstede, *Cultures and Organizations: Software of the Mind - Intercultural Cooperation and Its Importance for Survival*. McGraw-Hill, 1997.

[IEEE, 1987] *IEEE Guide to Software Configuration Management*, The Institute of Electrical and Electronics Engineers, 1987.

[IEEE, 1990] *IEEE Standard Glossary of Software Engineering Terminology*, The Institute of Electrical and Electronics Engineers, 1990.

[IEEE, 1998] *IEEE Standard for Software Configuration Management Plans*, The Institute of Electrical and Electronics Engineers, 1989.

[ISO, 1995] *ISO 10007: Quality Management - Guidelines for Configuration Management*, International Organization for Standardization, 1995.

[Karolak, 1998] Dale Walter Karolak, *Global Software Development: Managing Virtual Teams And Environments*. IEEE Computer Society, 1998.

[Kelly, 1996] Marion V. Kelly, *Configuration Management: The Changing Image*. McGraw-Hill Book Company, 1996.

[Leblang and Chase, 1987] D. Leblang and R. Chase, Parallel software configuration management for networks, *IEEE Software*, **4**, 6 (November 1987), 28-35.

[Leblang and Levine, 1995] David B. Leblang and Paul H. Levine, Software configuration management: Why is it needed and what should it do? In: *Software Configuration Management: selected papers / ICSE SCM-4 and SCM-5 Workshops*, 53-60.

[Leon, 2000] Alexis Leon, *A Guide to Software Configuration Management*. Artech House, 2000.

[MacKay, 1995] Stephen A. MacKay, The state of art in concurrent, distributed configuration management. In: *Proceedings of the 5th International Workshop on Software Configuration Management*, 180-193.

[McLaughlin, 1996] Mark McLaughlin, Tackling the problems faced by geographically dispersed development teams. In: *Proceedings of the 2nd Joint Conference on AUUG 96 and Asia Pacific World Wide Web*, 1996**.**

[MIL, 1994] *MIL-STD-498 Software Development and Documentation*, The Department of Defense, 1994.

[Mockus and Herbsleb, 2001] Audris Mockus and James Herbsleb, Challenges of global software development. In: *Proceeding of the Seventh International Software Metrics Symposium*, 182-184.

[Mordechai, 1994] Ben-Menachem Mordechai, *Software Configuration Management Guidebook*. McGraw-Hill, 1994.

[Moreira, 1999] Mario E. Moreira, The 3 software configuration management implementation levels. In: *Proceedings of the 9th International Symposium of System Configuration Management SCM-9*, 244-254.

[Murugesan, 1999] San Murugesan, Leverage global software development and distribution using the Internet and Web. *Cutter IT Journal*, **12**, 3 (March 1999), 57-61.

[Nokia, 2001] Nokia Networks, Internal document, December 2001.

[Paulk *et al.*, 1995] Mark C. Paulk, Charles V. Weber, Bill Curtis, Mary Beth Chrissis, *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley Publishing Company, 1995.

[Pressman, 1997] Roger S. Pressman, *Software Engineering A Practitioner's Approach*. McGraw-Hill Companies, 1997.

[Ramesh and Dennis, 2002] V. Ramesh, Alan R. Dennis, The object-oriented team: lessons for virtual teams from global software development. In: *Proceedings of the 35th International Conference on System Science*, 18-27.

[Rochkind, 1975] M. J. Rochkind, The source code control system. *IEEE Transactions on Software Engineering*, **SE-1**, 4 (December 1975), 364-370.

[Thompson, 1997] S. M. Thompson, Configuration management - keeping it all together. *BT Technology Journal*, **15**, 3 (July 1997), 48-60.

[Tichy, 1985] W. F. Tichy, RCS - a system for version control. *Software - Practice and Experience*, **15**, 7 (July 1985), 637-654.

[White, 2000] Brian White, *Software Configuration Management Strategies and Rational ClearCase - A Practical Introduction*. Addison-Wesley Publishing Company, 2000.

[Whitehead, 1999] J. Whitehead, The future of distributed software development on the Internet. *WEB Techniques*, **4**, 10 (Oct. 1999), 57-63.

[Whitgift, 1991] David Whitgift, *Methods and Tools for Software Configuration Management*. John Wiley & Sons, 1991.