Software Technology in a Networked Computer Environment: Configuration Management in Software Design and Development Teams

H. Tellioglu
Institut für Gestaltungs- und Wirkungsforschung, Abteilung für CSCW
Technische Universität Wien
Argentinierstrasse 8/187, A-1040 Vienna, Austria
e_mail: hilda.tellioglu@tuwien.ac.at

Abstract

The software design and development require cooperation and communication among all the professionals involved in the processes. Some activities are carried out individually, and some are usually collaborative tasks, such as the establishment of an overall system architecture, the distribution of system functionality and of activities, and the responsibilities within the design and development processes, the definition and implementation of sub-system interfaces, the testing and maintaining the implemented software applications, the finding out of better design and development methods and supporting tools. This paper is based on case-studies of work practices in two software development teams. The main focus is on development as a cooperative activity. In particular configuration management (CM) builds the main focus of the investigations as a method (and a tool) to support different phases of software design and development. CM running on a networked computer environment is concerned with creating consistent descriptions of how different parts of a software modules fit together, with controlling bug reports, change requests and design changes, with creating and managing different versions of a system.

1. Introduction

At the beginning I will specify a framework for summarizing the different stages of software design and development and the actors involved in these phases. The activities of several persons involved in cooperative work must be carried out and articulated in terms of who, what, where, when, how, with which result, by means of what, etc. ‘Actors’ (participants in the cooperative work whose activities are being articulated) engaged in cooperative work of software design and development teams are managers, analysts, designers, programmers, testers, support staff and customers. They all are involved with the actual software design and development ‘tasks’ (operational intention, goals, obligations and commitments to meet) comprising certain ‘activities’ (unfolding course of purposive action) [3], [5], [8], [9].

CM is a cooperative task and needs technical and organizational support. The different kinds of CM methods and tools, and the role of CM in software design and development in a networked environment are described in the first part of the paper. The second part includes the description of the cases focusing on differences and commonalities of design and development practices, and specially the work environment regarding the use of CM tools.

2. Software Design and Development

The studies by Simone and Schmidt of articulation of cooperative work provide detailed insight into the workings of articulation work [13]. In a software design and development environment there is also articulation work with its different dimensions. ‘Mechanisms of interaction’ are defined by Simone and Schmidt as a set of symbolic artifacts that stipulate and mediate the articulation of the distributed activities - symbolic because of the necessity to manipulate the mechanism independently, and artifacts because it must be available independently of any particular situation. At the same time, the symbolic artifacts reduce the complexity of articulation work.

To understand the varying procedures both in software design and development, it is necessary to differentiate between design and development of software. The first activity required for building and verifying of software is software design following the analysis and specifications of software requirements [2], [9], [15]. Each technical step produces information in a manner that finally results in validated computer software. Software requirement transforms information to feed the design activity which produces a data design, an architectural or hierarchical design, and a procedural design¹. The software design

¹ Data design forms information domain model built during analysis into the data structures. Architectural design defines the relationship among structural components of the software. Procedural design transforms structural components into a procedural description of the program.
determines the quality of the software, and is therefore very important. Coding coupled with debugging and testing builds the *software development* part.
2.1 Actors and Tasks

The **manager** of a group can be a first-level manager supervising - that’s why they are often called supervisors - the programmers, designers, and others involved. In software design and development environment there are usually two types of managers: top managers and team leaders. In two software houses of our case studies people from top management were involved mainly in raising money for further development of intended software that is in both cases standard software and not a special software for a specific customer. Team leaders were completely involved with the work their people are doing and were easily able to take over a piece of the job if necessary. Behind spontaneous designing of program modules, helping the programmers on demand and sometimes programming the complicated program codes, some of the team managers tried to solve the financial problems of the company which is gone bankrupt after five years. In the most of the time team leaders were technical leaders finding answers to occurred problems by discussing them in the group. In an ideal case there must be no doubt that technical leaders totally understand people’s output in the group and are competent to judge its quality, but in our case studies there were often conflicts in the group because of the lacks in management work of the team leaders. Depending on the scarcity of time, which occurs because the deadlines are given by customers or by trade fairs or because it takes too long time to develop even the very easy parts of software, team leaders were not able to act as promoters and communicators making communication possible between the group members and to find the easiest solutions to the existing emergency problems. Team managers were situated in the organizational hierarchy between top managers and programmers transferring the requirements and changes in requirements from top management and marketing department to programming group. Sometimes they had co-ordination and communication problems among themselves or with top management which was also transparent to the developers and caused additional unlike situations in the organization.

Small software houses like ones in our case studies with 20 - 30 people are usually not able to afford specific persons who work only as analysts or as designers - who build a bridge between the analysts and the programmers - and do nothing else. In all case studies we made programmers (also team leaders) are analysts and designers at the same time. At the beginning of the project they are involved with learning about the customer and his proposed system by way of reading, interviewing and/or participating. After understanding the information domain for the software, as well as the required function, performance and interfacing, they write down a description of what it is the customer wants, get customer management to sign a contract and write the document in such a way that it is accurate, complete, unambiguous, understandable, implementable, and tell what the (software) system has to do without describing about how to do. Then they change to designers and translate the what documents (problem or requirements specifications) into how documents which should be a natural extension of the problem specifications.

The next actor in software design and development is the **programmer** who translates a design specification into a machine-readable form writing the software code and documents, debugging and testing them. As mentioned above, programmers are usually analysts and designers at the same time. New ideas for programs were created during informal conversations among programmers. If a programmer has a problem to which he or she does not know the answer he or she goes to the next colleague explaining the actual problem. Sometimes the occurred problem - whether a design or development problem - must be analyzed and discussed in the whole group. The initiator of the discussion schedules an emergency meeting immediately. He or she represents the problematic issue and other help him/her by the solution telling him/her the way to solve it or whether it is possible to solve it in one discussed way without considering the details of its implementation.

The **tester** can be everyone such as analyst, designer, programmer, manager, customer and tester. The individual programmer tests his or her module (unit testing) which focuses on the logical internals of the software ensuring that all statements have been tested. This can be controlled by setting some standards and providing some advice and support through involvement by the manager or supervisor or co-workers. During the programming phase programmers execute the integration tests whose planning should be done during the design phase and early programming phase. Program system testing is usually performed by people other than the programmers. They test the finished program system from the customer’s point-of-view. This test focuses on the functional externals to conduct tests to uncover errors and to ensure that the program produces required results.

The **customer** often executes acceptance test and site test. During the test phases everything must be tested such as the modules and the entire program, documents like proposal, problem, design and coding specifications and user manuals. The **support people** can work in fields such as secretary, quality assurance, library, etc. The work of that people usually is more obviously of a service nature.

Behind this named tasks there is one more task to illustrate - maintenance. After delivering the produced software to the customer several changes will occur. That can have a lot of reasons. The customer probably requires functional or performance enhancements. Errors have been encountered. The delivered software must be adapted to accommodate changes in its external environment.
2.2 Configuration Management in Software Design and Development

Software designing includes the configuration of the software system [7]. To determine the system’s organization and construction, the identification of all the modules that make up the system, the description of each module in the system and description of the interdependencies of the modules in the system are necessary. The other elements of system’s configuration are the description of how the modules fit together and the documentation of changes made to each module. Behind the various types of tools used in software development, configuration management takes a very important place within software developing process. It is not the most important aspect of software development, but for medium- and large-scale projects it is a practical necessity. It goes without saying that configuration management contains common resources such as information and material resources, in the terms of mechanisms of interaction.

Compared to such terms as ‘coordination’, ‘cooperation’ and ‘control of process’, the concept of configuration management provides a number of benefits in a CSCW context: First, documentation change control is required in all phases of the software design and development. Second, configuration management offers a lot of functions such as controlling the software system’s organization, construction and maintenance. Third, configuration management is needed to prevent: high software maintenance costs due to unreliable products; systems that are improperly constructed due to the inclusion of one or more out-of-date modules; systems in the field that are incompatible with each other; inability to upgrade software in order to meet new requirements; two or more programmers making conflicting changes to the same module; programmers making unauthorized code changes; lack of coordination between quality assurance personnel and programmer; disasters when programmers leave the project or their duties are assigned to someone else.

Forth, without configuration management, it is often the case that only one person knows the information regarding which modules are needed to build the system, how they depend on each other and the processes required to incorporate all the modules into the system. If this person leaves the project, much of this valuable information can be lost. Fifth, configuration management also provides the ability to determine the differences between two different revisions of a module, to revert to a prior revision of a module and to determine when a particular bug was introduced into the system.

The following figure shows the links between the software design/development process and the configuration management.

![Diagram showing links between software design/development process and configuration management](image)

**Figure 1.** Configuration management and software design/development process.

There are significant differences among the various CM tools [1]; the tool can be file- or object-oriented, it can represent changes physically or logically, and it can apply changes sequentially or selectively. There are two main groups of CM models:

- The **sequential model** supports the building of tools for archiving and retrieval of previously named versions and not for really managing the software system. The version is the fundamental unit of specifications. Differences between an old version and its successor represent changes. Files are used to save the changes physically, and therefore the system is

---

2 ‘Revision’ is used to refer to a new form of a module. ‘Version’ shows the new form of the entire software system. A system version can be thought of as a slice across the various revisions of the system’s component modules, including one revision from each module.
limited to the kinds of information available in files (e.g. the names and the contents of files). The system can retrieve only the versions that are read into it. The basic type - the 'linear sequential model' - does not provide a good method of dealing with parallel versions, whereas the 'branched sequential model' allows developers to work with parallel versions through branching and merging. A tree of versions is created which can be retrieved by the system any time.

- CM tools based on the selectable-change model are more than simple version-retrieval systems. They are object-oriented and therefore a change saved logically as an object can be associated with whatever information is meaningful to the developer. With this tool new combinations of prior changes can be selected to produce new versions; an old version of the file (e.g. a source code or an object file) can be compared to an altered version and applied independently. This is made possible by characterizing each version (implicitly or explicitly) by a list of changes to be administered. The change builds the fundamental unit of specifications. Any combination of changes can be used to specify a version.

A configuration management system contains a set of tools fulfilling different functions. First, a version control system controls changes to source code. It documents the history of changes made to each source code module and allows to revert to a prior revision of a module if necessary. It also documents who made each change, what the change involved, and when the change was made. Second, object module librarian manages object codes. It deals with code after it has been compiled and is used to organize the code into object libraries. A primary benefit of using libraries is that they eliminate the need to specify explicitly every single object module required to build an application. A good librarian promotes structured programming by emphasizing modularity. Once modules have been defined, created and tested, related modules can be placed in a library. Then other programmers can refer to these modules in their code and the linker will automatically integrate these referenced modules from the library into the final program. Such an approach emphasizes module reusability. Third, make utility is an automatic system builder. A text file called a ‘makefile’ identifies all the modules that make up a software system, indicates how the modules depend on each other, and what procedures must be performed to build the system.

Software configuration management tools are somewhat related to collaborative authoring systems reported by McAlpine and Golder [4]. The concept of a collaborative authoring system is to support synchronous and asynchronous communication between authors involved with writing documents together and supporting the writing process which contains of five stages building an iterative model: coordination, writing and annotation, consolidation, negotiation and coordination. If all authors have agreed on a version of the document they have written, the iteration discontinues.

To describe the software development process of a software development team it is necessary to show the structure of the software product which must be located on a central unit (so called ‘server’ in terms of client/server architecture) in the networked environment so that all members of the programming team may access the files. The programmers are connected to each other and to the server. The program files on the server must be divided into directories based on their function, i.e. the modules of the program build functional subsystems where each subsystem has its own directory. These program modules include source files (for instance C files), object files (usually with .obj extension) and eventually other miscellaneous files such as makefiles. In the structure of the entire program there are also other subsystems such as library files not necessarily located in separate subdirectories. These libraries include functions used by the program modules. Therefore, library files are linked to the program by the makefile which compiles the source files of the modules, creates object files and links all together as well as the libraries. In short, makefiles build the software system at that level. There are also makefiles at the level of each programmer, i.e. each module can be compiled and linked eventually with the other necessary modules by each programmer locally in order to debug and test the written code. If no errors can be found, the programmer can ‘put’ the tested module into the server. The following figure shows through an example the structure of the modules and the entire program on a server of a networked development environment.

---

3 DEC’s Code Management System (CMS), Intersolve’s PVCS and Mortice Kern Systems (MKS) are some examples of tools based on the sequential model. PVCS will be described thoroughly within the scope of case studies in the following sections.

4 There are some CM tools based on selectable-change model, such as Aide-de-Camp, IBM Update, CDC Update and Historian Plus.
The steps which are to be taken during developing a complex software program can be described depending on the size of the software project and on the number of designers/developers working on the same project. The different project sizes can be distinguished in four categories [10]:

- **Small single-person projects** contain usually at most 25 files configured in a single directory containing all of the source and object files, makefiles and other miscellaneous files. One developer is involved in this kind of projects. He or she checks out (gets) files with a lock in order to modify them and checks them in (puts) again. Locking can be used in this situation as a security measure, and prevents the checking in of experimental changes.

- **Medium-to-large single-person projects** include usually more than 25 files which are grouped into directories on a functional basis. Logfiles and source files are separated through locating in subdirectories. Locking and write protection are used to maintain the security. One configuration file governs the operation of CM for all project directories.

- **Multiple-person, non-networked projects** present the greatest difficulties for the project manager. One central machine can be used as a central database. Each developer has to go to the central machine to check in or out the files. This can be done via batch files. The copy of files is done by using floppy disks where a lot of consistency problems can occur.

- **Multiple-person, network-based projects** are also structured in directories based on the functions of files. The project directory structure and the configuration files are located on the network file server so that all developers can have access to the files. Each developer duplicates usually the project directory structure on his or her own workstation. Because each developer generally modifies only a small number of the files at any time, he or she usually maintains only files being changed on his or her individual workstation. The object code of the other modules can be accessed on the network server.

The multiple-person, network-based projects are relevant for this paper and will be discussed in the following sections. The interactions between the actors regarding the use of the CM tools are complex and needed to be described with one very common example: What happens if one developer wants to modify a source file which is locked by another one in the team? On the one hand a lot of communications between the two developers and on the other hand some operations on the network server are necessary in such a situation. The following figure displays the necessary steps to handle this kind of situations.

1. Developer 1 locks and gets the file A in order to modify it.
2. Developer 2 tries to get the file A, but the access is denied, because the file A is locked.
3. Developer 2 contacts the developer 1 and requires a branch of the file A.
4. Developer 1 puts the file A back to the server and unlocks it.
5. One of the developers or the project manager creates a branch of A (for simplicity reasons the files are here called A1 and A2).
6. Both developers lock and get their corresponding files.
As mentioned above, each module put onto the server has a revision number. A program system gets a system version as a slice of the various revisions of the modules. A module can be accessed by its revision number. Behind the possibilities to access the modules separately of the whole software program, it is also possible that two or more different programmers work on the same module at the same time. At that point two functions of a configuration management tool must be introduced: branching and merging. Branching enables management of more than one path of code development. A branch is a development path that diverges from the primary path of software development. While one works along the primary development path, other programmers can create their own branches to work on their individual tasks. All information on branch revisions is maintained in a single log file. After branching, a merge must be done to two or more different sets of changes into a single new revision of the module. Any conflicting changes will be marked so that the programmer can resolve the conflict. In sum, configuration management tools allow more than one programmer to make changes to the same module. To make it clear to understand, the following figure illustrates an example of branching and merging of files [10].

Figure 3. An example to access the same file at the same time with a CM tool in a networked software development environment.

After all file changes one developer or the project manager who is responsible to create new program versions requires at a certain point of time that all developers put their program modules back onto the server (step (1)). All developers put their files on the network server and unlock them (step (2)). He or she then locks all files of the software project and creates a new program version. Afterwards he or she unlocks the files again to further development (step (3)). The following figure illustrates the creation of a new program version.

Figure 4. An example of branching and merging of files with CM tools.
3. Cases

In the two case studies standard qualitative research methods such as semi-structured interviews were used to explore the design and development practices in groups especially regarding the use of CM tools. Participant observation could not be made because both organizations do not exist since a couple of years [14].

3.1 The First Case

The organization of the first case study consisted of two managers, one organizing assistant and one developer at the beginning. During the following years new developers were employed to accomplish the increased complicated programming work. At the end there were eight developers, five people in management and a few professionals for user documentation (e.g. manuals) and for public relations. Participant observation could not be made because both organizations do not exist since a couple of years [14].

The choose of the working environment, of the newest system and network software, and of the standard application development kits was made by the developers themselves, especially by the chief developer. To use the best possible development environment including both software and hardware had always the most priority. Development tools were homogeneous in order to make it possible that each programmer could work on each computer any time. As the development environment was networked the source code, object files, resource files and other miscellaneous files are located on the server. The only thing a programmer had to do was to get his or her program modules from the server and to make a local test environment. At the beginning the developers had problems by introducing the PVCS as CM tool. Some of the developers generated documents about the use of PVCS by explaining the most important features of the tool. This helped the others in the team by understanding the necessity and the way how to use it during developing the software. PVCS enabled the transparency of program modules and the easy management of different versions of the whole program (managing of actually developed code, tested code, code of a program version which is already installed by a customer etc.). On the one hand, each programmer had the possibility to look at the code of the others (code review), to make explicit remarks to some of them mostly via e-mail, to change parts of that code, to learn new methods and to improve his or her programming skills. On the other hand, mistaken code structures and procedures or uncommented functions and program modules were also obvious through the used CM tool to everyone in the group.

Research of new programming styles, aims to improve the developed program both in its structure and performance, learning new programming tools, styles and languages was strongly encouraged mainly by the chief programmer who was the project leader of the development group and through him also by the management. Although the company had serious financial problems there were always enough money to buy books, new software or to visit an important programmers’ meeting or conference.

At the same time there were a lot of conflicts because of the different priorities and values between developers and management. The vision of the developing group which included being perfect programmers and developing the best standard software in that field could not always understood by the management very well. Because of the scarcity of resources managers wanted to offer a completed steady software as a solution for office automation, but developers needed
more time to fulfill the wishes of management. Several times management promised customers new program versions without talking about these with the developers to clarify the product status. The deadlines set by the management were ignored by the developers. For urgent functions or program versions developers took their time as they normally needed.

Task distribution happened during the meetings participated by the people of management as well. Among the professionals there was the possibility to express the favorite module that a programmer preferred to develop next. Each developer was specialized in one part of the whole program and had the responsibility of designing and developing that modules, building a subprogram, testing its functionality and its integration with the other subprograms and finally maintaining it. In the development group there was the tendency to create equal access possibilities to resources and knowledge. Senior developers were generally ready to help the newcomers answering their questions about complex design and development procedures of specific program modules.

One of the most important rules among the developers was writing explanatory comments both for each submodule, for each procedure in all modules and for each complex code part containing calls of other functions or complicated algorithmic structures.

### 3.2 The Second Case

The number of people occupied in the software house of the second case study varied between twenty and thirty depending on resources and status of the product development stage. In times when there were a lot of development jobs to do, approximately thirty people including both experienced professionals and students of computer science were engaged in developing a standard software on the platform of OS/2 operating system. The intended product was a software package for office work. There were two subgroups of developers working on the two separated parts of the product, namely the database and the user interface part including three subprograms, e.g. for calculation, for word processing and for drawing graphics. All these applications had to be combined to each other through communication tools, e.g. data could be transported from one application to another.

Incomplete specifications caused a lot of problems for the organization. After one year development was stopped because management and project leaders of technical departments did not know how to make the further design and development. In the first place they were very busy solving the financial problems they had. All students involved in development were unemployed. In a second phase the most students could restart their jobs. The in the first phase created code was not applicable and had to be changed in its functionality. In contrast to the first phase the developers were allowed to design their part in the second period. There were no methods applied in the entire company for design processes. Designers wrote down the specifications they found necessary for their modules in a way they used to. Sometimes, because of lack of time, they started to program without designing the program modules.

Deadlines for a new version was given by the managers who were not interested in problems and conflicts occurring in the subgroups. Local interaction within the subgroups (database and user interface) was strong. Sometimes there were real conflicts among developers having different views of programming styles and how to behave in a team. Personal relationships within the small subgroups became more important supported through informal communications which was used for expressing and discussing problems occurring during software design or implementation. Ad hoc meetings were used as places for changing ideas and considerations about the applications. In contrast to wellfunctioning local connections, communication to the other groups did not even exist. No one from the both groups had an overview of the existing software. Each developer wrote their own subprocedures. Therefore there were several versions of the same functions. A coordinator went once through the development department asking programmers about their modules and functions. He tried to document everything he founded via charts for enabling an overview of functions developed in both teams, and to distribute that information to everyone in the company. This was not efficient enough to create a transparency of the work done in the company. Management tried to find solutions for communication problems mainly between two subgroups. They arranged special days as ‘organizational meeting days’ where everyone was supposed to explain their activities. Those meetings were held twice a year, but seen ineffective and meaningless by most developers. Both groups working in completely separated rooms saw each other only in the official meetings once a week. For instance, developers of user interface group did not even know the names of the database functions and their parameters they had to use in their own modules. Each time new functions were developed by the database group, there were conflicts between both groups deflecting blame to each other: Who was responsible for unplanned implementation of program modules? Whose responsibility was initiating design meetings before developing any modules? Why did not work the communication between both groups? Management tried to organize a research due interviews with all developers about reasons and status of the communication between the both subgroups. During the second phase of the organization, a network was used to manage software and to enable communication between people in the company. Some of them used e-mail to communicate
different topics with each other, private or organizational. Some of them used the network only for getting new software from the server and eventually to put own developed software onto it.

Introducing the networked environment in the organization caused a lot of troubles for everyone. OS/2 as operating system could not be run with network software at the same time on the same computers. The software development stopped for a week, because all computer systems had crashed after running the network software. The result was that everyone was involved within the installation of the network. Afterwards all software was located on the server. The developers started to use PVCS as a CM tool. They got their files to work on them and tested the modified code in the standardized testing environment on their own computer.

Until integration of both subprograms (database and user interface) both groups worked with dummy functions and aimed at finishing their own part. Unrecoverable errors, system crashes and inconsistent software was the result of first integration of both subprograms. To their surprise the developers recognized that there was a very important part, so called “data model”, missing between both subprograms. Without the data model programmers of the user interface group were forced to access the low level database functions directly in their applications and to use the same data structures as in database applications. Rearrangement of existing data structures and of “dummy functions” took too long (each time about a week).

In the first phase no one paid attention for the code documentation which could help the reuse of the created functions. In the second phase there were two types of documentation: source-inline documentation which consisted of comments about source code and about functionality of the module, and documents for user manuals written by the developers. In the second phase of development some of the programmers forced to find out errors in codes developed by others who were not in company at all. They had to correct them and add new parts to these modules.

In the first phase there were neither naming conventions nor notations for coding, where as the team leaders defined how to name the procedures, variables and fields of data structures, and implemented particular tool, i.e. a code editor with macros, for coding process which enabled for instance the setting of brackets or comments on a specific line and column. This tool which enabled a standard programming style only within the groups.

4. Conclusion

In the first case through PVCS used as a CM tool, a shared view of the system and a high level of transparency were created. A rotational principle ensured that all developers were involved in maintaining and updating the overall structure of the system. The developers tried in general to value and enforce the best practices and the disciplined use of robust naming conventions as well as filling the module headers with detailed descriptions. This facilitated the mutual understanding of each others’ modules. The chief developer put a considerable amount of work into creating clear dependency definitions between modules and components and into visualizing them in graphic form. There was always an overview of the current project status - versions that were already completed and implemented, versions ‘in design’, versions that were tested by an external tester. Changes made by developers were documented and transparent to all in the development team, because PVCS allowed to represent changes as complete source files and as files of differences to source files. The shared development environment was supported technically through a network and organizationally through flexible meetings. Changes and program errors were visible to all in the development team. All team members could easily manage their modules within the other modules. Program errors were often discussed within the team during meetings, where those who had created an error were also responsible for fixing it. Developers supported each other. Team members generally did not feel unqualified, because the programming mistakes were perceived as normal. The developers were open to discuss the code complexity, the occurred errors, and methods to solve the problems. The PVCS’ standard ‘make utilities’ were used by all developers to build a entire software system. Everyone in the development team could be the integrator of the software system. At a certain time only one developer was responsible for managing the versions and for creating the software system. Each developer was able to produce their revisions. There were procedural guidelines using PVCS as a memorandum created by one of the developers and updated by others in the team if necessary. Important decisions about creating new versions and branches were made within the team. As there was a good overview of existing versions (via PVCS), the architecture and the structure of the software system, it was not difficult to reproduce old versions of the software product.

In the first phase of the second case, versions were simply created by copying the files of different programmers from one computer to another using floppies. It was impossible to create an overview of the whole system and both team leaders had difficulties in creating a structured product through (re)designing modules and dependencies. During the software integration period the whole development process collapsed. When it was restarted, both subteams introduced PVCS. Only the two team leaders were informed about the current project status. The use of PVCS was limited. Both subgroups did not want to discuss and generate shared practices and to use them in a disciplined way. Control of changes was not well
developed in the first phase. A platform for performing a system test could not be created. The integration of completed subprograms was not possible. The introduction of PVCS helped greatly in making changes and program errors visible to all within the subgroups. The developers used different batch files and varying switches for compiling and linking the modules. These caused the system inconsistencies which were visible during the software integration period. Because no one had an overview of existing and developing program versions, and they did not use a network and a CM tool to manage the software system, it was impossible to reproduce old program versions in the first phase. In the second phase developers recognized the necessity of a networked computer environment, of ‘make utilities’, and of a CM tool. These proved helpful in supporting program integration. The lack of cooperation and communication between the two subteams (database and user interface) was reflected in the incomplete definition of the interface between them.

Configuration management is a very important process during software design and development in a networked environment. Especially if the project is a multiple-person network-based project, it goes without saying that the development environment has to be a networked environment with supporting communication and coordination services (like e-mail, scheduler). Applications supporting the software configuration (like CM tools) are essential to keep track of the tasks in the development team and to maintain the software versions and dependencies between the modules. It is also necessary that the developers understand the importance of such tools and use them regularly to insure the consistency of the software product.

5. References


