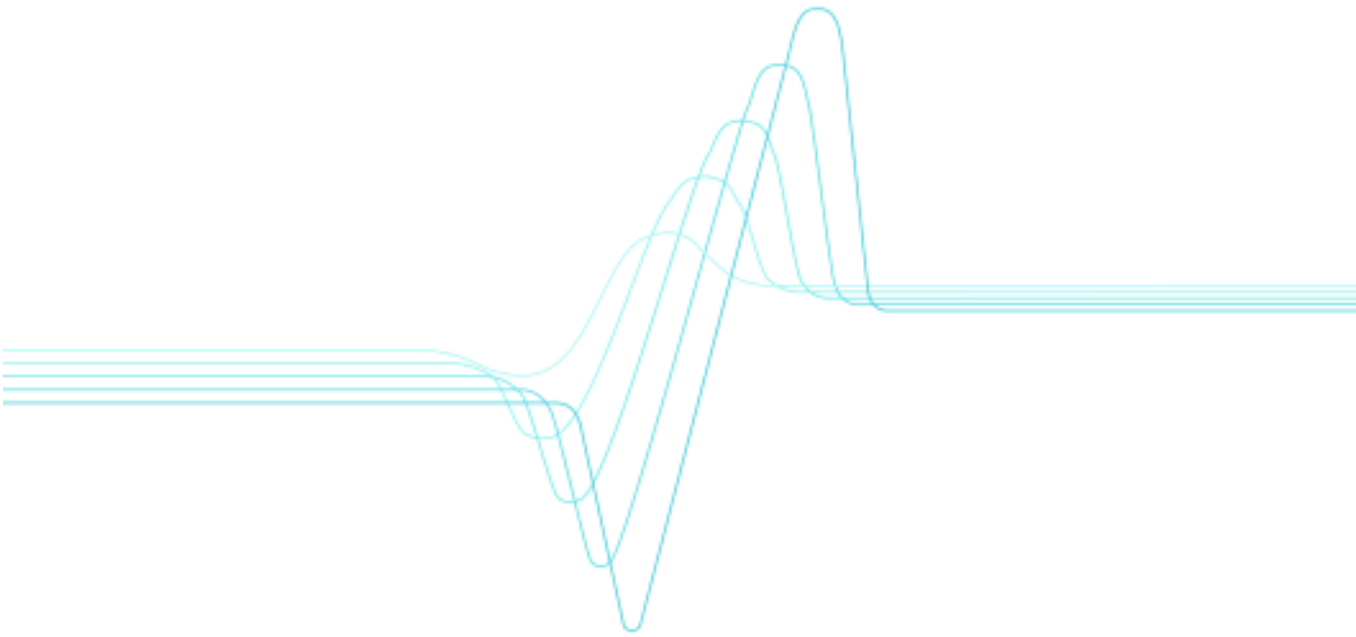
ESPOO 2003 VTT PUBLICATIONS 514

Juha Koskela

Software configuration management in agile methods



VTT PUBLICATIONS 514

Software configuration management in agile methods

Juha Koskela

VTT Electronics

ISBN 951–38–6259–3 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–6260–7 (URL: http://www.vtt.fi/inf/pdf/)
ISSN 1455–0849 (URL: http://www.vtt.fi/inf/pdf/)

JULKAISIJA – UTGIVARE – PUBLISHER

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

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

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

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

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

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

Technical editing Marja Kettunen

Otamedia Oy, Espoo 2003

Koskela, Juha. Software configuration management in agile methods. Espoo 2003. VTT Publications 514. 54 p.

Keywords: software configuration management (SCM), agile methods, extreme programming, software development methods

Abstract

The development of good quality software is a critical element of successful competition for today's software market shares. However, software products are becoming larger and more complex; therefore, the development of quality software is neither easy nor rapid. Agile software development methods focus on generating early releases of working products. They aim to deliver business value immediately from the beginning of the project. Regardless of the development method in use, it is important that software development be under control. Software configuration management (SCM) is known as a method of bringing control to the software development process, and thus, proper application of SCM is a key component in the development of quality software. However, currently very few studies on software configuration management in agile methods exist; hence this study.

The aim of this publication is to systematically review the existing literature on agile software development methodologies from an SCM point of view. First, analytical SCM lenses based on existing SCM literature are constructed. Second, existing agile methods are analyzed using the lenses constructed. The results show that only two of the existing agile methods take SCM explicitly into account, but most of the methods highly value SCM tool support and its ability to revert to earlier versions of development artefacts. Nonetheless, the basis for successful SCM implementation, SCM planning, has been completely forgotten.

Contents

List of symbols

CCB	Configuration Control Board
CI	Configuration Item
CM	Configuration Management
CSA	Configuration Status Accounting
CSCI	Computer Software Configuration Item
FCA	Functional Configuration Audit
PCA	Physical Configuration Audit
SCM	Software Configuration Management
SCMP	Software Configuration Management Plan
VSS	Microsoft Visual SourceSafe
VTT	Technical Research Centre of Finland (http://www.vtt.fi)

1. Introduction

Today software products are becoming larger and more complex. At the same time, stronger market pressures are forcing software engineers to develop their products more quickly. Furthermore, customers are demanding more with better quality, and requirements keep changing. These facts can lead to a failure if the software product development is not well under control.

In brief, software configuration management (SCM) is a method of controlling the software development and modifications of software systems and products during their entire life cycle (Crnkovic et al. 2003). SCM affects a product's whole life-cycle by identifying software items to be developed, avoiding chaos when changes to software occur, offering traceability for the changes made, by providing needed information about the development state, and aiding the audit of both software and the SCM process.

In the last few years, agile software development methods have gained significant attention in the software engineering community. Agile methods focus on generating early releases of working products and on delivering business value immediately from the beginning of a project. Agile software development methods contain minimal processes and are often labelled "barely sufficient" (Highsmith 2002a). On the other hand, SCM is often considered as bureaucratic method that causes additional work and more documentation. However, according to Compton & Conner (1994), SCM is essential whenever software is developed. SCM cannot be eliminated; at most one can achieve bad SCM (Compton & Conner 1994). Thus, regardless of the software development method used, the role of SCM remains important. Because there exist very few studies on software configuration management with agile methods, this study has been undertaken.

The goal of this publication is to systematically review existing literature on agile methods from an SCM point of view. First, analytical SCM lenses based on existing SCM literature are constructed, and second, existing agile methods are analyzed using the analytical SCM lenses constructed.

This work is organized in six sections. In the following second section, an introduction to software configuration management and the lenses for the

subsequent analysis are provided. The third section presents an overview of the existing agile methods, which are subsequently analyzed in section four using the analytical SCM lenses constructed. In the fifth section, results and implications of the study are discussed. The sixth section presents the conclusions of the study.

2. Software configuration management

In this section, software configuration management is introduced. Thus, the purpose of SCM, basic SCM activities, and the automation of SCM are explained, giving a picture of complete software configuration management. Finally, the construction of SCM lenses for the subsequent analysis is presented.

2.1 Background

Configuration management (CM) is the discipline of controlling the evolution of complex systems (Tichy 1988). CM first came into existence in U.S. defence industry (Leon 2000), where it was used to control manufacturing processes. Gradually, computers and software also evolved to the stage, where people were constrained to find ways to control their software development processes. In short, SCM is CM specifically for software product development. Tichy (1988) presents two ways in which SCM differs from general CM: First, software is easier and faster to change than hardware, and second, SCM can potentially be more automated. However, according to Abran & Moore (2001, p. 122), "the concepts of configuration management apply to all items to be controlled although there are some differences in implementation between hardware CM and software CM."

Today there are various standards for SCM, for both military and commercial use. According to Leon (2000), the most comprehensive international standard is ANSI/IEEE standard 1042 (1987, p. 10). It defines SCM as follows:

> *"Software CM is a discipline for managing the evolution of computer program products, both during the initial stages of development and during all stages of maintenance."*

2.2 The purpose and benefits of SCM

SCM is a critical element of software engineering (Feiler 1990). According to Leon (2000), it is needed because of the increased complexity of software systems, increased demand for software and the changing nature of software.

Leon also states that SCM can be used as a strategic weapon that will give the organization an edge over those who are not using SCM or using it less effectively. When used effectively during a product's whole life-cycle, SCM identifies software items to be developed, avoids chaos when changes to software occur, provides needed information about the state of development, and assists the audit of both the software and the SCM processes. Therefore, its purposes are to support software development and to achieve better software quality. As it can be seen in Figure 1, SCM is one of the major elements leading to better software quality.

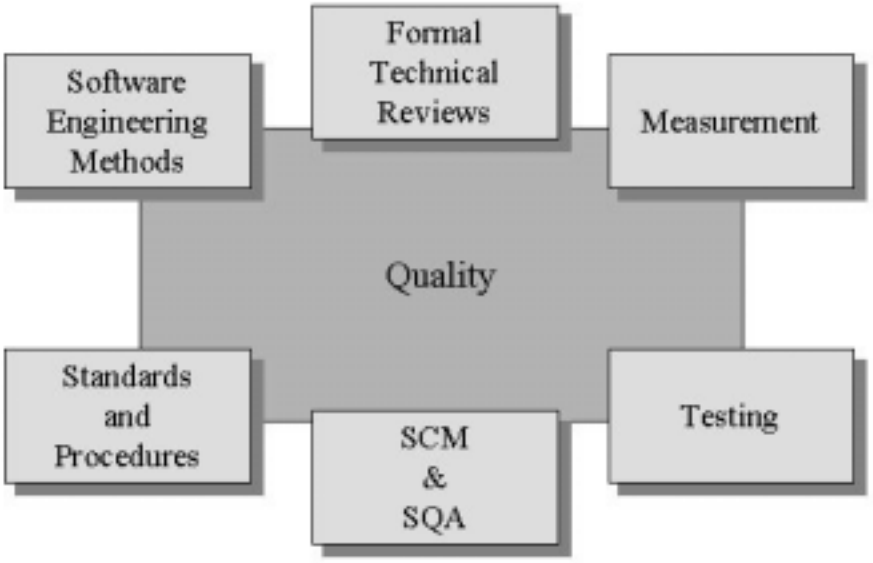


Figure 1. Achieving software quality (Pressman 1997).

2.3 SCM activities

According to the IEEE's (IEEE Std. 828-1990) traditional definition of SCM, the following four activities are included: configuration identification, configuration control, configuration status accounting and configuration audits. Successful SCM implementation also requires careful planning (Abran & Moore 2001). SCM planning produces a document called SCM plan, in which SCM activities

and other practices of SCM are described carefully (IEEE Std. 828-1990). Figure 2 shows these five basic SCM activities.

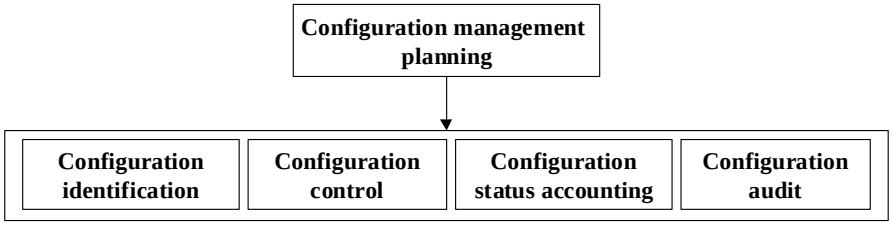


Figure 2. The basic activities of SCM.

2.3.1 Configuration identification

According to Leon (2000), configuration identification is a process where a system is divided into uniquely identifiable components for the purpose of software configuration management. These components are called computer software configuration items (CSCIs) or shorter and more generally just configuration items (CIs). A CI can be a unit or a collection of lower level items (Rahikkala 2000). IEEE (IEEE Std. 610.12-1990) defines configuration identification as an element of SCM, consisting of selecting the CIs and recording their functional and physical characteristics in technical documentation. Each CI must be named and versioned uniquely to distinguish it from the other CIs and from other versions of CIs (Whitgift 1991). Examples of CIs are project plan, specifications, design documents, source codes, test plans and test data, executables, make files, tools, and SCM plan. Whitgift (1991) also states that every source item should have a status attribute which defines the level of approval that the item has achieved. An example of the range of status values for an element code is: untested, module tested and integration tested. Accordingly, a document can have such values as draft, proposed and approved.

In the configuration identification phase, a project's baselines and their contents are also identified. A baseline is a software configuration management concept that helps us to control change (Leon 2000). It is a document or product that has been formally reviewed and that thereafter serves as a basis for further

development. It can also be an assembly of CIs, an accepted configuration (Taramaa 1998.) The most common baselines of software development are shown in Figure 3 below (Pressman 1997). The life cycle model shown is a traditional waterfall model; every phase produces a baseline:

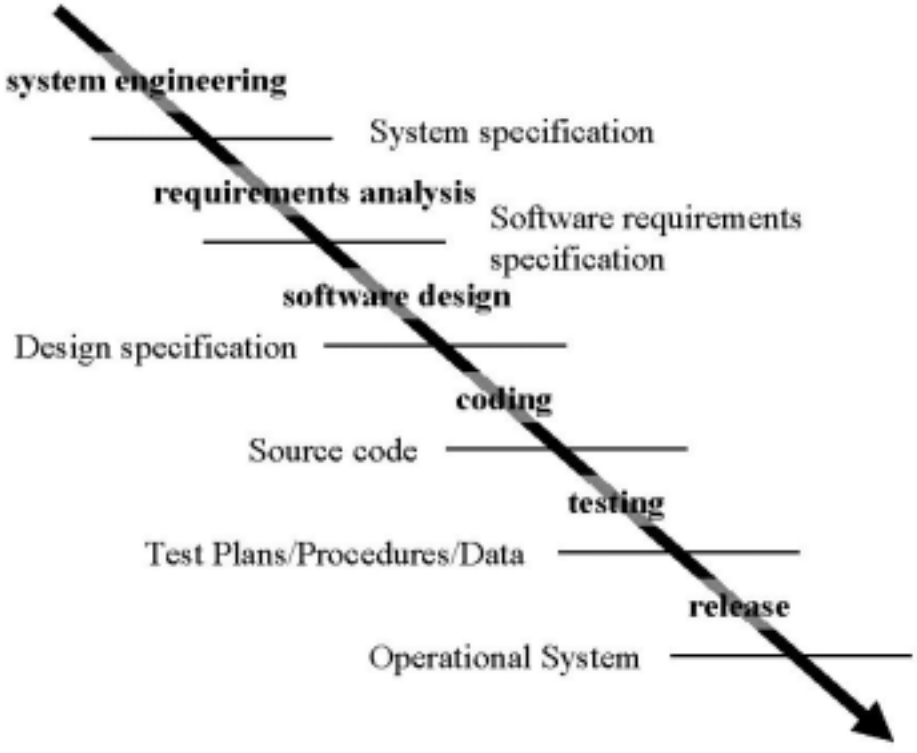


Figure 3. The most common software baselines (Pressman 1997).

2.3.2 Configuration control

As stated earlier, software can change very quickly and easily, and uncontrolled changes can lead to chaos. Therefore, after the configuration items of the system have been identified, the next step is to control the changes to the software.

Controlling changes during software development has been defined as a task for SCM (Pressman 1997). According to Leon (2000), baselines have a very important role in managing change. According to (IEEE Std. 610.12-1990),

baselines can be changed only through formal change control procedures including the following steps: evaluation, coordination, approval or disapproval and implementation of changes to configuration items.

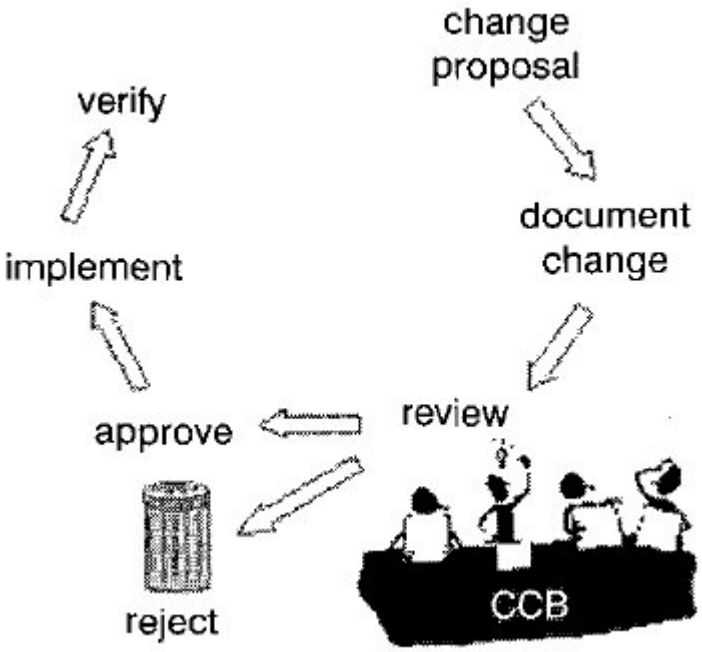


Figure 4. The change management process for conventional software (Asklund & Bendix 2002).

A change request can result from many things. For example, new features, enhancements of existing features or defects can lead to change requests (Leon 2000.) Figure 4 above presents a traditional change control process. The process starts, when a need for change is noticed. A properly completed change request form is sent to the configuration control board (CCB), whose main function is to evaluate and to approve or disapprove change requests (Leon 2000). According to Leon (2000), change requests can also be deferred when they and their associated documentation are filed for later resolution. If a change request is approved, proposed changes are assigned to developers to be implemented. After implementation, the changes are verified in various testing procedures to ensure that they have been implemented as agreed.

Change is one of the most fundamental characteristics in any software development process (Leon 2000). Lehman (1980) also suggests that change is intrinsic in software and must be accepted as a fact of life. Making changes to software is easy, but if it is done at will, chaos will result (Leon 2000). According to McConnell (1998), effective projects control changes, whereas ineffective projects allow changes to control them. However, Whitgift (1991) states that the level and formality of control required varies considerably; large teams need strict and formal change control, but small teams can rely on much less formal control.

2.3.3 Configuration status accounting

Software development produces lots of information that should be recordable and reportable whenever needed. According to IEEE (IEEE Std. 610.12-1990), configuration status accounting consists of the recording and reporting of information needed to manage a configuration effectively, including a listing of the approved configuration identification, the status of proposed changes to the configuration and the implementation status of approved changes. All this and other information related to CIs and activities concerned with them are thus available for the people involved in the project. Status accounting reports include change logs, progress reports, CI status reports and transaction logs (Leon 2000)

2.3.4 Configuration audits

According to Leon (2000), the purpose of configuration audits is to ensure that the software product has been built according to specified requirements (Functional Configuration Audit, FCA), to determine whether all the items identified as a part of CI are present in the product baseline (Physical Configuration Audit, PCA), and whether defined SCM activities are being properly applied and controlled (SCM system audit or in-process audit). A representative from management, the QA department, or the customer usually performs such audits. The auditor should have competent knowledge both of SCM activities and of the project (Leon 2000).

2.3.5 SCM planning

According to Abran & Moore (2001), a successful SCM implementation requires careful planning and management. All of the SCM activities introduced above are described in the SCM plan. The main purpose of the SCM plan is to answer such questions as: who is going to do what, when, where, and how (Buckley 1996). Thus, the SCM plan serves as a guideline for the people working with software configuration management. According to Buckley (1996), a configuration management plan is written for each project. However, an organization may use a generic SCM plan template that can be tailored to each particular project. Table 1 presents the common contents of an SCM plan according to IEEE standards:

Table 1. Common contents of SCM plan (IEEE Std. 828-1998).

Class of information	**Description**
Introduction	Describes the Plan's purpose, scope of application, key terms, and references.
SCM management	(Who?) Identifies the responsibilities and authorities for accomplishing the planned activities.
SCM activities	(What?) Identifies all activities to be performed in the project.
SCM schedules	(When?) Identifies the required co-ordination of SCM activities with the other activities in the project.
SCM resources	(How?) Identifies the tools and physical and human resources required for execution of the Plan.
SCM plan maintenance	Identifies how the Plan will be kept current while it is in effect.

As it can be seen from Table 1, an SCM plan defines responsibilities, activities, schedules and resources related to a project's SCM implementation.

2.4 Automation of SCM

The basic SCM activities introduced in the previous subsection are assisted by SCM tools. According to Leon (2000) the role of an SCM tool is to automate SCM activities and to improve development productivity. Automating manual SCM tasks provides more time to do the actual development work, leading to improved speed and productivity. From a developer point of view, SCM offers a stable development environment, maintains configuration items, stores their history, supports product building and coordinates simultaneous changes (Crnkovic et al. 2001). In other words, it helps software developers in their daily work. According to Estublier (2000), a typical modern software configuration management tool provides primary services in the following areas:

- Management of the repository of components
  - Version control
- Support for engineers
  - Workspace management
  - Concurrency control
  - System building
- Process control and support

These features are explained in the section 2.4.2. In the next section, the automation of SCM activities is introduced in more detail.

2.4.1 Automating SCM activities

According to Dart (1990), SCM is generally a combination of manual and automated procedures. However, since most SCM processes are well established and repetitive, they can be supported by computers (Conradi & Westfechtel 1999). We know that configuration identification is an SCM activity in which configuration items are selected and their functional and physical characteristics are recorded in technical documentation. According to Leon (2000), this type of information can be captured and updated automatically by an SCM tool.

Change management is an activity that is supported in many of today's SCM tools (e.g. Conradi & Westfechtel 1999; Leon 2000). As one example, information about change requests goes directly to all the people concerned (such as the CCB), and they can then send approval or disapproval immediately by e-mail or other messaging systems. All the information related to the change process, such as who initiated the change, who implemented the change, and how the change was implemented, can be captured and used for status accounting to manage the whole project more effectively (Leon 2000.) This and other relevant information can be then queried for the purposes of various reports.

As explained earlier, configuration auditing is the validation of the completeness of a product. According to Leon (2000), SCM tools can automate most of the auditing, because they can generate the necessary information for verification purposes. For example, one person might need a history of all changes and another a log containing details about work completed.

2.4.2 Common features of SCM tools

Today there are a number of SCM tools available, while their features vary greatly. According to Leon (2000, p. 202), "each tool has its own strengths and weaknesses. For example, some are better at change management, whereas others have excellent build management and versioning capabilities." Next, these features are examined in more detail.

Version Control

The main purpose of version control is to manage different versions of configuration objects that are created during the software engineering process (Pressman 1997). According to Taramaa (1998), the term “object” is used by some authors to describe CI. A “component repository” is a place where all the versions of CIs are kept. The terms “version”, “revision” and “variant” are basic SCM concepts. Therefore, their differences are explained next. The meaning of “branching” and “merging” is also explained.

Let us assume that a developer is producing an item not yet under SCM control. S/he can freely modify the item, and the changes made affect this item, in particular. But when the item is put under SCM control, changes to the item produce new revisions, CIs that change and evolve with time. They may be created for various reasons, such as to extend functionality or to fix bugs (Conradi & Westfechtel 1996).

Variants are versions that are organized in parallel development lines called branches. However, branches are only one way to implement variants, because they can be done also with conditional compilation, with installation description, or with a run-time check (Persson et al. 2001). There are two types of variants: permanent and temporary (Whitgift 1991). Permanent variants are created, for example, when the product is adjusted for different environments. There can be a variant for Linux and another for Windows. The difference between a permanent and a temporary variant is that a temporary variant will later be integrated with another variant (Zeller 1996). This kind of integration is also called “merge”. According to Whitgift (1991), an instance of an item, each of its variants and revisions, is called an item version.

Figure 5 presents a situation where development is done in parallel branches. When revision 1.3 is almost done, revision 1.2, which is already in use at a customer site, is noticed to have a bug. If there is no time to wait for revision 1.3 to be approved, a temporary bug-fix branch is created, the bug is fixed, and the variant is supplied to the customer. After that versions 1.3 and 1.2.2 are merged into version 1.4. At the same time, development is also done in a permanent variant branch for Linux.

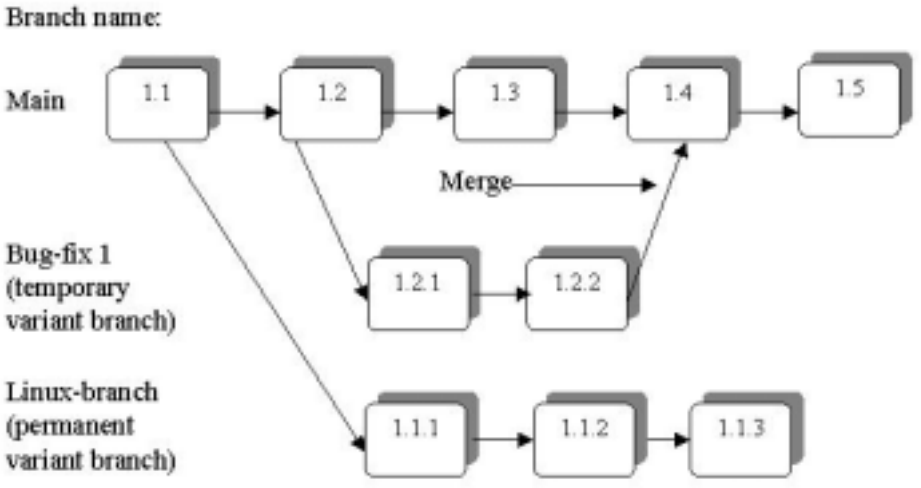


Figure 5. Basic version control.

Version attribute values, mentioned in the SCM process section, are also supported by many CM tools (Whitgift 1991). They can provide facilities for recording the evolution of an item through its lifecycle.

Workspace management

SCM tools can also help developers to work in peace by providing them with private workspaces. The purpose of provided workspace is to prevent users from interfering with one another's work (Dart 1994). Developers can check out desired files or configuration from the repository in their own workspaces. The "check out" operation creates copies of the items in the developer's workspace; after that, the developer can modify the items as needed and finally execute "check in". The "check in" operation copies items back into the repository: the version control system then creates new versions as needed. Figure 6 represents the check out/in process. A "get" operation equals to a "check out" operation, except that the files copied into the workspace are read-only and therefore cannot be modified. The terminology in use for these operations can vary between different SCM tools.

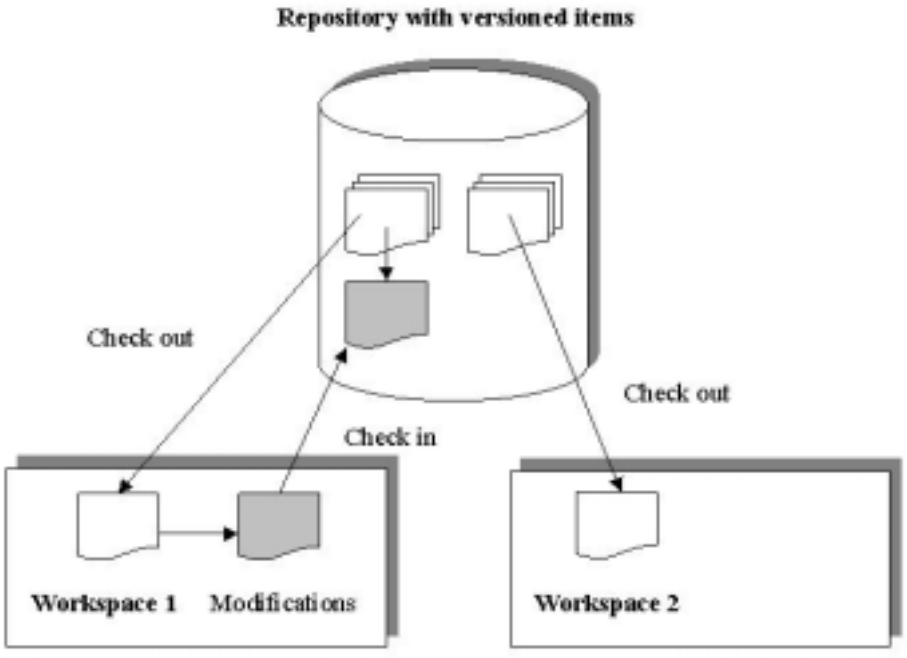


Figure 6. Workspaces and check out/in.

Concurrency control

The implementation of concurrency control depends on the SCM tool in use. Some of these tools use so-called optimistic concurrency control (Asklund & Bendix 2002), which means that files are not locked when checking them out, and there may be simultaneous modifications to the same files by multiple users. Others prevent simultaneous access by using a locking mechanism. Microsoft's Visual SourceSafe (VSS) is a one example of an SCM tool in which both of these mechanisms are implemented and locking is used as a standard setting. Figure 7 below presents a situation where one user is trying to check out a project file already checked out by another user; the tool's locking mechanism prevents this operation by disabling the check out function.

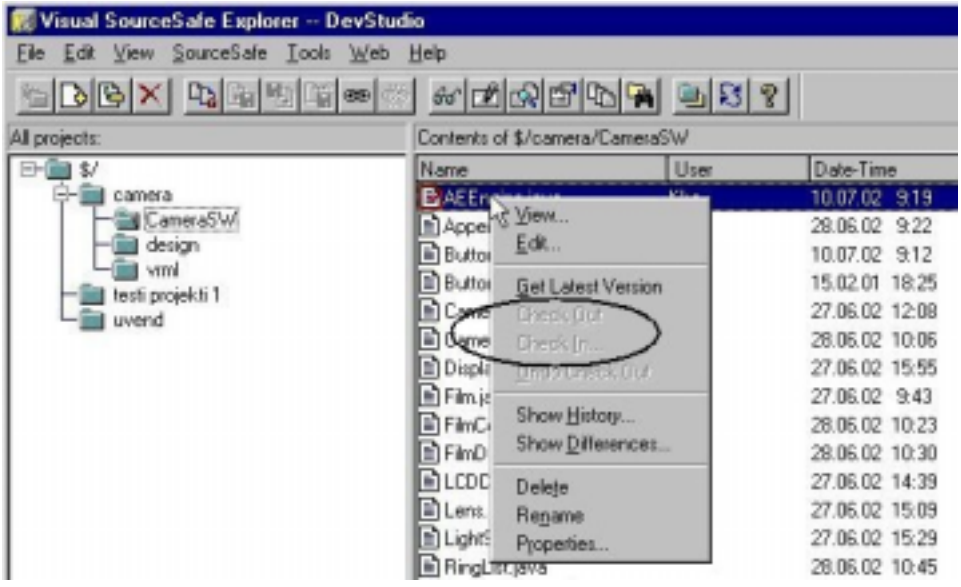


Figure 7. Preventing simultaneous modifications by using file-locking mechanism in VSS.

However, if the “multiple checkouts” function is enabled in VSS, users are able to check out files that have already been checked out by another users. The users checking in such files must themselves resolve any conflicts between modifications that may occur if VSS is not able to merge the modifications automatically. Figure 8 presents this situation, where the users have modified the same lines, and therefore the user who checks in later has to solve the problem. All activities should be based on the project's SCM procedures as defined in the SCM plan.

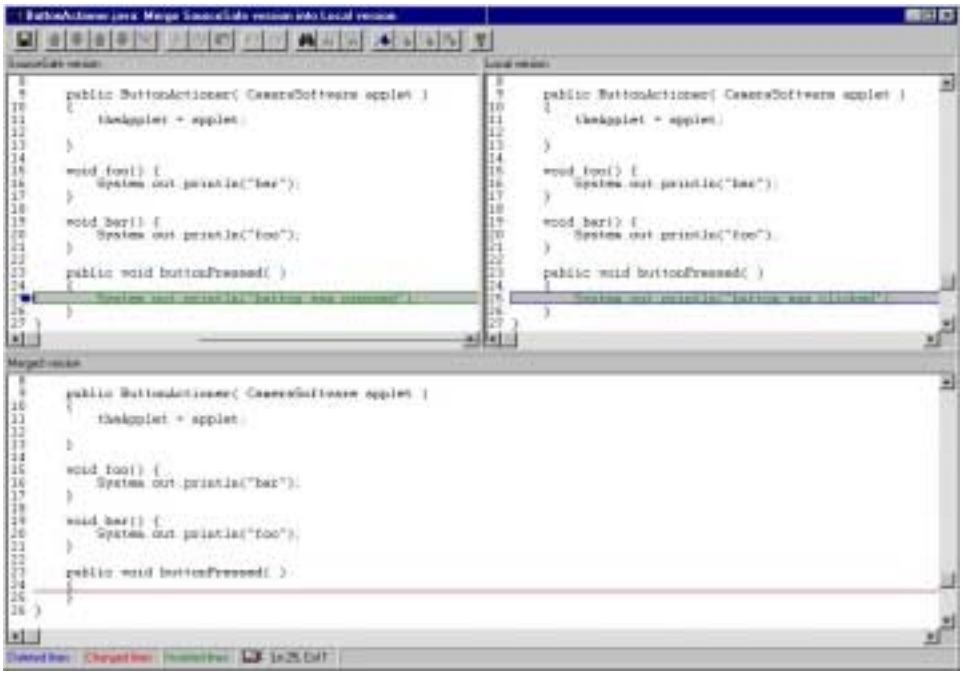


Figure 8. Merging the overlapping modifications in VSS.

The advantages of concurrent development are remarkable. Development teams can work with the same files, while the SCM tool controls development. According to Leon (2000), the difference between concurrent and parallel development is that in concurrent development, branches are finally merged into a single item, whereas in parallel development merging is not performed and branches go ahead separately. The same alternatives can exist at the project level. For example, if there is a project that reaches the testing stage, the project can be branched so that testing and bug fixing can be performed separately. Development continues in the main branch, and later on the modified files on these two branches can be merged.

System building

In system building (a.k.a. build management), the purpose is to combine needed file versions and then compile them to create an application. Building can be done for a subsystem, a module, or a whole system (Leon 2000) and the building strategy can be either incremental or from scratch. In incremental building, only

files that been changed will be built; correspondingly, all files will be built when building from scratch. According to Whitgift (1991), building from scratch can be a very expensive process in terms of machine resources, because a large system may contain thousands of files of source code.

SCM tools can facilitate building work by capturing necessary information and making the building process repeatable. To be repeatable, details of used components, components' versions, the operating system and its version, compiler and linker options etc., are required (Leon 2000). An SCM tool usually includes a Make tool to generate an executable software system reliably and efficiently (Conradi & Westfechtel 1998).

Process control and support

According to Estublier (2000), process support means both the "formal" definition of what is to be performed on what (a process model) and the mechanisms to help reality conform to the model. In practice, a State Transition Diagram (STD) and activity-centred modelling are offered as a solution. As an example Estublier (2000) states that change control is traditionally an integral part of SCM tools.

2.5 Lenses for the analysis

This section constructs the SCM lenses through which current agile software development methods will be analyzed. The following analytical lenses were seen as relevant for addressing the research purposes of this publication; see Table 2.

Table 2. Lenses for the analysis.

Perspective	**Description**	**Key references**
Software configuration management approach	How is software configuration management addressed in the method?	(Feiler 1990; Leon 2000)
SCM planning	What does the method define as SCM planning?	(IEEE Std. 828-1990; Abran & Moore 2001; Buckley 1996)
Configuration identification	What does the method define as item identification and baselining?	(IEEE Std. 828-1990; Leon 2000)
Change management	What does the method define as change management?	(IEEE Std. 828-1990; Leon 2000)
SCM tools	What is the role of software configuration management tools in the method?	(Leon 2000)

As has been said, software configuration management is a critical element of software engineering (Feiler 1990). According to Compton & Conner (1994), it is an essential activity that must take place whenever developing software. SCM is needed because of increased complexity of software systems, increased demand for software and the changing nature of software (Leon 2000). Again, according to Leon (2000), proper application of SCM is a key component in the development of quality software. This perspective explores how the method in question addresses SCM. For example, the method may have SCM as a key practice; then its viewpoint towards SCM can be seen as explicit. Therefore, this perspective also explores the various methods' SCM-related practices.

According to Abran & Moore (2001), a successful SCM implementation requires careful planning. *SCM planning* is the basis for project's SCM implementation precisely recording in an SCM plan who is going to do what, when, where, and how (Buckley 1996). This viewpoint examines what the method defines as SCM planning.

According to Leon (2000), *configuration identification* is the basis for subsequent control of the software configuration. It is "the process whereby a system is separated into uniquely identifiable components for the purpose of SCM" (Leon 2000, p. 91). From this perspective, what the method in question defines as item identification and baselining will be analyzed.

As stated before, change is one of the most fundamental characteristics in any software development process (Leon 2000). Lehman (1980) states that change is intrinsic in software and must be accepted as a fact of life. According to Leon (2000), making changes to software is easy, but managing those changes (the uncontrolled changes) is not. However, uncontrolled change can create problems serious enough to create project failures. Controlling changes while software is being developed has been defined as a task for SCM (e.g. IEEE 828-1990; Taramaa 1998). This viewpoint explores agile software development methods from a *change management* perspective. In other words, what the method in question defines as change management and how it should be handled will be examined.

The purpose of *software configuration management tools* is to support and automate SCM activities and to provide help for developers. According to Leon (2000), the role played by SCM tools is becoming more and more important in today’s complex software development environments. Leon (2000) also states that no SCM tool is the solution for every software configuration management problem, but SCM tools can be a step towards more effective software configuration management. Weatherall (1997, p. 3) sums this up as follows: "SCM is first an attitude; second, a process; and only third, a set of tools." However, software configuration management tools are important part of a comprehensive software configuration management solution and, therefore, they can be seen as relevant to the evaluation of agile software development methods from the SCM point of view. This dimension explores what the role of SCM tools is in a particular agile method.

3. Current state of agile software development methods

In this section, agile software development in general and existing agile software development methods in particular are introduced briefly.

3.1 Overview of agile software development

The agile movement has gained significant attention in the software engineering community in the last few years . Agile software development methods focus on generating early releases of working products. It is claimed that they are good at adapting to change. In fact, according to Cockburn & Highsmith (2001a), agility is “all about” creating and responding to change.

The name "agile" arose in 2001, when seventeen process methodologists held a meeting to discuss future trends in software development. They noticed that their “lightweight” methods had many characteristics in common, and as the term “lightweight” was not descriptive enough for these methods, they agreed on the term “agile”. According to Cockburn (2002), an agile process is both light and sufficient. “Lightness” is a means of staying manoeuvrable. “Sufficiency” is a matter of “staying in the game”. In consequence of this meeting, the "Agile Alliance" and its manifesto for agile software development (Beck et al. 2001) emerged. The manifesto states the following:

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions *over processes and tools*

Working software *over comprehensive documentation*

Customer collaboration *over contract negotiation*

Responding to change *over following a plan*

That is, while there is value in the items on the right, we value the items on the left more."

As it appears of this manifesto, agilists value things on the left more. However, the idea is to find the right balance between these items. For example even if the working software is valued more by agilists, they can not forget documentation completely.

People have been recognized as the primary drivers of project success in agile methods (Cockburn & Highsmith 2001b). People are the most important ingredients of success, because good and rigorous process and the right tools will not save the project if the development team does not have skilful members (Martin & Hall 2001). With the “right” people, the ability to communicate with other team members is also valuable and helps to keep things up to date.

Working software without documentation is better than non-working software with volumes of documentation. In agile methods, working software is delivered early and often. It is valued more than comprehensive documentation, and therefore documentation can be added later when there is time. But what is the right amount of documentation? According to Cockburn (2002), words "just enough" and "barely sufficient" can also be applied to documentation.

The third statement values customer collaboration over contract negotiation. Working closely with the customer and the frequent delivery of software enable regular customer feedback, which is needed for successful projects.

The last statement of the manifesto values responding to change over following a plan. Since change is inevitable in software development (Pressman 1997), developers must be able to react to change when it happens. Plans are important, but the problem is that software projects can not be accurately predicted far into the future, because there are so many variables to take into account (Martin & Hall 2001).

3.2 Shared characteristics

Agile software development methods have some common characteristics. As these methods are "barely sufficient", the two main characteristics are, as indicated before, lightness and simplicity. Agile methodologies thus contain minimal processes and documentation and reduced formality (Highsmith 2002a). The purpose is to do what is needed at this moment, not to try to predict the future too far ahead, and to do urgent things first. Fast and incremental development enables fast partial “delivers” to customers and further fast feedback from them. Abrahamsson et al. (2002) enumerate four elements that make a development method agile. In this case software development should be incremental, cooperative, straightforward and adaptive; see Table 3:

Table 3. Common characteristics of agile methods (Abrahamsson et al. 2002).

Common characteristics of agile methods	**Description**
Incremental development	Small software releases, with rapid cycles.
Cooperative	Customer and developers working constantly together with close communication.
Straightforward	The method itself is easy to learn and modify, and it is well documented.
Adaptive	Able to take into account last moment changes.

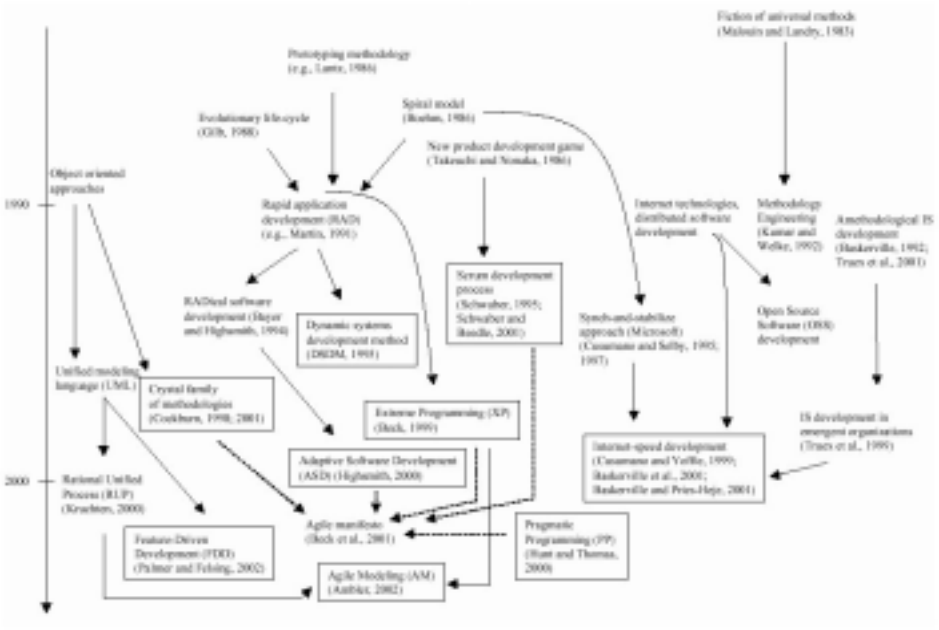


Figure 9. Evolutionary map of agile methods (Abrahamsson et al. 2003).

Figure 9 above shows agile methods and their interrelationships, together with their evolutionary paths. Abrahamsson et al. (2003) have purposefully included some more philosophical meta-level expositions in the figure, because these have either directly or indirectly impinged upon preceding agile methods. Figure 9 shows that most of the current agile methods appeared near the year 2000.

3.3 Existing agile methods

The family of agile methodologies is growing constantly. According to Abrahamsson et al. (2003) and their comparative analysis of agile software development methods, nine methods existed when they wrote their article. Seven of these methods, excluding Agile Modeling (AM) and Internet-Speed Development (ISD), are included in this analysis. AM is excluded, because it is not method *per se* (Abrahamsson et al. 2002): AM requires supporting methods, as it covers only modelling. Respectively, ISD is excluded, because very few articles in it exist. Today there also exist other agile software development methods, such as Lean Software Development (Poppendieck & Poppendieck

2003), but they are still reasonably new to be included in the analysis. As a result the included methods are Adaptive Software Development, the Crystal family of methodologies, Dynamic Systems Development Method, Extreme Programming, Feature Driven Development, Pragmatic Programming and Scrum.

This section briefly introduces existing agile methods. Because of the research interests of this publication, the purpose is not to introduce agile methods thoroughly and comprehensively. For a more extensive review and analysis of current agile methods, readers are referred to (Abrahamsson et al. 2002).

3.3.1 Adaptive Software Development

Adaptive Software Development (ASD), developed by James A. Highsmith, offers an agile and adaptive approach to high-speed and high-change software projects. The method encourages incremental, iterative development, with constant prototyping. The static plan-design-build life cycle is replaced by a dynamic speculate-collaborate-learn life cycle (Highsmith 2002b).

According to Highsmith (2002b, p. 310), "The practices of ASD are driven by a belief in continuous adaptation." However, ASD's practices are difficult to identify, and that is perhaps the most significant problem with the method (Abrahamsson et al. 2002).

3.3.2 Agile Modeling

Agile Modeling is a chaordic, practice-based methodology for effective modeling and documentation of software-based systems (Ambler 2002a). According to (Abrahamsson et al. 2002, p. 82), "The underlying idea is to encourage developers to produce advanced enough models to support acute design problems and documentation purposes, while still keeping the amount of models and documentation as low as possible." AM is not a complete software process; the focus is primarily on effective modeling and documentation (Ambler 2002b).

3.3.3 Crystal family of methodologies

Crystal is the name of a family of methodologies developed by Alistair Cockburn. The Crystal family focuses on individuals' talents and emphasizes that every team should utilize a process uniquely tailored to itself (Highsmith 2002a). As the focus is on people, interaction, and community, it means that processes and tools are secondary. According to Highsmith (2002a), there are only two absolute rules in Crystal: incremental development and self-adaptation. In a Crystal project, increments should be less than four months, and the methods and conventions the team adopts must be tuned and evolve.

3.3.4 Dynamic Systems Development Method

According to Stapleton (1997, p. xiv), "DSDM is a framework of controls for RAD, supplemented by guidance on how to apply those controls." DSDM provides a user-centred, iterative and incremental way of developing application systems that serves the needs of the business (Stapleton 1997). Since its origin in 1994, this method has become the *de facto* standard for RAD in the UK. DSDM is based on nine principles, all of which must be applied in a project; otherwise the whole basis of DSDM is endangered (Stapleton 1997).

3.3.5 Extreme Programming

Extreme programming (XP), developed by Kent Beck, is an agile method for teams developing software subject to rapidly changing requirements. It is a deliberate and disciplined approach to software development (Crispin & House 2003). The XP process can be characterized by short development cycles, incremental planning, continuous feedback, reliance on communication, and evolutionary design (Beck 1999b). Further according to Beck (Beck 1999a), rather than planning, analyzing, and designing for the far future, XP suggests doing all these activities a little at a time, throughout the software development. Primarily, XP is aimed at object-oriented projects using at most a dozen programmers in one location (Jeffries et al. 2001). According to surveys, XP is currently the most popular and best-known method in the agile family of methodologies (e.g. Maurer & Martel 2002; Charette 2001). XP is made up of a

simple set of common-sense practices. In fact, most of XP ’s practices have been in use for a long time and therefore they are not unique or original (Beck 1999a). Many of them have been replaced with more complicated practices in the process of time, but in XP they are collected together.

3.3.6 Feature Driven Development

Feature Driven Development (FDD) was first reported in (Coad et al. 2000) and further developed by Jeff Luca, Peter Coad and Stephen Palmer (Abrahamsson et al. 2002). FDD is a highly iterative agile software development process that is focused on delivering frequent, tangible and working results. The FDD approach does not cover the entire software development process, but rather focuses on the design and building phases (Palmer & Felsing 2002). The method is built around a set of "best practices", which should be used to get the greatest benefit from the method. According to (Palmer & Felsing 2002), the practices are not new but the particular combination of the ingredients is new.

3.3.7 Internet-Speed Development

According to Abrahamsson et al. (2003), Internet-speed development is the least known of agile software development methods. Baskerville et al. (2001) have researched the ISD approach and present three important aspects related to it: “time-drives”, “quality depends” and “process adjusts”. “Time drives” means that ISD development is focused on moving the product quickly to market. By “quality depends” the authors mean that quality depends on how important it is to customers and how good the developers are. The third one means that the development process must adjust according to current needs and circumstances.

3.3.8 Pragmatic Programming

Pragmatic Programming is actually a set of programming "best practices" published in "The Pragmatic Programmer" by Andrew Hunt and David Thomas (2000). According to Abrahamsson et al. (2002), Pragmatic Programming does

not include process, phases, distinct roles or work products. However, the book contains a total of 70 tips that cover most programming practicalities.

3.3.9 Scrum

Scrum is a management and control process that focuses on building software that meets business needs (Schwaber & Beedle 2002). According to Highsmith (2002a), it has also been applied to nonsoftware projects. Scrum has a project management emphasis and deals primarily at the level of the team. In fact, the name Scrum refers to the game of rugby, because of the similarities between these two; both are adaptive, quick and self-organizing (Schwaber & Beedle 2002). According to (Schwaber & Beedle 2002), Scrum teams are fully autonomous and guided by their knowledge and experience, rather than formally defined project plans. Scrum does not define any specific software development techniques or methods for the implementation phase, but rather it leaves the power of decision to the developers. However, Scrum introduces a set of practices that establish an environment within which products can be rapidly built in complex environments. Schwaber & Beedle (2002) suggest strict adherence to the practices, if the team is novice to Scrum. Later, when the level of experience has grown and people have adopted the values of Scrum, it is possible make adjustments.

4. Results

The purpose of this section is to analyze chosen agile methods from an SCM point of view. Analysis is conducted using the analytical lenses defined in section 2.5. Thus, for each method the software configuration management approach, SCM planning, configuration identification, change management approach and the role of SCM tools is evaluated. Each perspective will be analyzed separately in the following subsections.

4.1 SCM approach

Table 4 shows how different agile methods address software configuration management and enumerates the methods' SCM related practices. The methods take very different approaches towards SCM. If a method takes SCM explicitly into account, it indicates that the value of SCM has been clearly understood and emphasized. Therefore, the methods with an explicit approach clearly require SCM to be conducted during the project. If a method has a tool viewpoint to SCM, it indicates that that the literature about the method deals with software configuration management from a tool perspective. If a method does not explicitly take SCM into account, software configuration management can still be used to support software development with that particular agile method.

FDD and DSDM take software configuration management explicitly into account. In FDD, SCM is one of the method's "best practices" that among others needs to be included in order to obtain the greatest benefit from the method. In addition, one FDD practice suggests building a system regularly, for example, daily or weekly, depending on the size of the project and the build time (Palmer & Felsing 2002). According to Palmer & Felsing (2002), an FDD project's demands on a CM system depend on the nature and complexity of the software under development. Besides FDD, DSDM also values SCM highly. According to Stapleton (1997, p. 61), "software configuration management is a key feature in DSDM." Good configuration management is essential in order to control the rapidly evolving products within a DSDM project. One of the DSDM's practices requires that all changes during the development be reversible.

Table 4. Software configuration management approach and SCM related practices.

Method	**Software configuration management approach**	**Practices related to SCM**
Adaptive Software Development	SCM not explicitly considered	–
Crystal family of methodologies	Tool viewpoint on SCM	–
Dynamic Systems Development Method	SCM explicitly considered	All changes during the development must be reversible.
Extreme Programming	SCM partially considered	Collective ownership, small releases, and continuous integration.
Feature Driven Development	SCM explicitly considered	Configuration management, regular builds.
Pragmatic Programming	Tool viewpoint on SCM	Source code control.
Scrum	SCM not explicitly considered	–

XP's approach to SCM can be seen as implicit. The method does not require software configuration management to be conducted. However, Paulk (2001) has examined XP from a CMM (Capability Maturity Model) perspective and states that SCM is partially addressed in XP via collective ownership, continuous integration and small releases. The Crystal family of methodologies and Pragmatic Programming have a tool perspective towards software configuration

management. This means that the literature of those methods deals with SCM only from a tool perspective.

In both ASD and Scrum, SCM is not explicitly addressed. However, in the implementation phase of Scrum, developers are free to use any methods and techniques they prefer to accomplish the aim of the Sprint. Thus, the use of software configuration management practices and tools is not excluded. Nonetheless, according to Abrahamsson et al. (2003, p. 6), "ASD is more about concepts and culture than software practices."

4.2 SCM planning

The planning of software configuration management should be the basis for a project's SCM implementation. According to Leon (2000, p. 152), the main purpose of the SCM plan "is to create an awareness among the various groups involved in a software project about the SCM functions and how they are to be performed in that project."

It was found that none of the agile methods addresses SCM planning specifically. However, it may be assumed that in FDD and DSDM, which require SCM to be conducted, SCM planning is intended to be part of a comprehensive SCM solution.

4.3 Configuration identification

From this perspective, it will be analyzed what each method defines as item identification and baselining. Table 5 shows that many of the methods stipulate that every development artefact should be under version control. These managed development artefacts are called configuration items in SCM terminology.

Table 5. Configuration identification.

Method	Item identification and baselining
Adaptive Software Development	Not explicitly addressed.
Crystal family of methodologies	Not explicitly addressed.
Dynamic Systems Development Method	Everything should be under version control. Baselining should be conductable daily.
Extreme Programming	Source codes should be under version control.
Feature Driven Development	Everything should be under version control.
Pragmatic Programming	Everything should be under version control.
Scrum	Not explicitly addressed.

DSDM suggests that "everything that is produced (analysis models, designs, data, software, tests, tests results, etc.) should be kept in step with one another so that it is relatively easy to move back to a known 'build' state whenever the development has gone down a blind alley." (Stapleton 1997, p. 73) FDD has a similar viewpoint as it suggests that "any artefact that is used and maintained during the development of the system is a candidate for version control." (Palmer & Felsing 2002, p. 53). Pragmatic Programming emphasises making sure that "*everything* is under source code control – documentation, phone number lists, memos to vendors, makefiles, build- and release procedures, that little shell script that burns the CD master – everything" (Hunt & Thomas 2000, p. 88). The literature of XP values the management of source code (Jeffries et al. 2001). Collective code ownership and continuous integration combined to source code control is the basis for XP development. Other agile methods do not explicitly define what items should be managed. DSDM is the only method that sets requirements for baselining requiring that it should be conductable daily (Stapleton 1997).

4.4 Change management

According to Cockburn & Highsmith (2001a), agility is about creating and responding to change. This section analyses agile methods from a change management perspective; see Table 6.

Table 6. Change management.

Method	**Change management**
Adaptive Software Development	Changes are frequent in software development. Therefore, it is more important to be able to adapt to them , than it is to control them.
Crystal family of methodologies	Not explicitly addressed.
Dynamic Systems Development Method	All changes during development must be reversible. Development team members must be empowered to make decisions without bureaucratic overheads. Change control serves to ensure that quality, once built in to the system, is preserved.
Extreme Programming	The customer decides what to change. The team can integrate its changes with current code through comprehensive unit testing.
Feature Driven Development	New requirements and features should be controlled by a change management process.
Pragmatic Programming	Not explicitly addressed.
Scrum	"Controlled chaos": Changes are managed using Product Backlog and Sprint Backlogs.

In FDD and DSDM, change management is a natural part of development, because they both require software configuration management to be conducted during a project. FDD requires that new requirements and features should be controlled by a change management process (Palmer & Felsing 2002). This change management process should include reviewing the impact on cost, schedule, and quality. For the resulting system to meet its expectations, the development process must allow controlled changes, deletions and additions to requirements and features. Changes to new features should be tracked separately from the main features list. Palmer & Felsing (2002) introduce the 10% Slippage Rule they have found useful in controlling changes. This rule means that a project can absorb a maximum of 10 % increase in features without affecting the project. In FDD this absorption is possible because the features are small and the duration of feature's development is short. Changes to the project beyond 10%, however, will force a 10% change in something else, for example, schedule or scope. DSDM requires that all changes during development must be reversible (Stapleton 1997). If a fruitless path has been taken during development, it is possible to revert to earlier stages of the development and proceed from there. Moreover, DSDM requires that development team members must be empowered to make decisions without bureaucratic overheads (Stapleton 1997). Further according to Stapleton (1997), the main function of change control in DSDM is to ensure that quality, once built into the system, is preserved.

In XP the customer decides what to change and can change the requirements (user stories) in the project, but they can not be changed during the iteration (Martin 2002). XP team members are empowered to make decisions during the iteration and the customer is always on-site to help clarify matters. The team integrate their changes continuously with currently released code through comprehensive unit testing. According to collective ownership practice, anyone can change any code at any time.

According to Schwaber & Beedle (2002), in Scrum the Product Backlog is a prioritized list of features, enhancements and bug fixes that constitutes the changes to be made to the product in future releases. Only one person, called the Product Owner, is responsible for managing and controlling the Product Backlog. Changing the contents of Product Backlog or priorities is done with the Product Owner. A Sprint Backlog is a list of tasks the team has to complete to meet the Sprint goal. However, no changes to the Sprint Backlog are allowed

during the Sprint. According to Highsmith (2002a), this is the "control" part of "controlled chaos."

According to (Highsmith 2000), most management strategies are geared either to reduce the number of changes or to control changes. From ASD's perspective, these strategies are useful, but it is more useful to embrace change than to try to control it. "Balancing at the edge of chaos" balances between change control and change containment (Highsmith 2000). By “change containment” Highsmith means tackling a problem and moving on.

In Pragmatic Programming and Crystal family of methodologies, the change management has not been explicitly addressed. As Pragmatic Programming does not include process, phases, distinct roles, or work products, it is understandable that the change management point of view has not been considered.

4.5 SCM tools

As mentioned above, the role played by software configuration management tools is becoming more and more important in today’s complex software development environments (Leon 2000). In this section agile methods are analyzed from the SCM tool perspective, see Table 7.

Most of the agile methods emphasize the importance of the SCM tool. Agile methods consider the ability to revert to earlier versions of development artefacts highly valuable. Quick changes may lead to ”a blind alley” in the development, and then it is important that every earlier version of every artefact is accessible. In fact only ASD and Scrum do not mention SCM tools at all. However, in Scrum, developers are free to use whatever methods and tools they prefer to accomplish the aim of the Sprint.

Table 7. The role of SCM tools.

Method	The role of SCM tools
Adaptive Software Development	SCM tools are not explicitly addressed.
Crystal family of methodologies	The SCM tool is one of the most important tools a project team can have.
Dynamic Systems Development Method	The role of the SCM tool is important. However, it has been understood that a tool does not solve anything by itself.
Extreme Programming	The importance of SCM automation is emphasised.
Feature Driven Development	SCM tool is required.
Pragmatic Programming	SCM tool is required.
Scrum	SCM tools are not explicitly addressed.

According to Cockburn (2002), in Crystal methods, versioning and configuration management tools are "the most important tools the team can own." (Cockburn 2002, p. 203) Unfortunately the method does not describe anything else about SCM or SCM tool usage. As mentioned earlier, DSDM requires a lot of configuration management. According to Stapleton (1997), every development artefact should be co-ordinated so that it is easy to revert to earlier versions. Therefore, Stapleton sees automated software configuration management as an essential part of the DSDM project. In addition to version control, the chosen SCM tool should support baselining and branching. Integration with other development tools and ease of use are highly valued. As referred to earlier, Stapleton (1997) also states that an SCM tool, in itself, does not solve anything; a tool is only as good as its users.

In XP, code changes are continuously integrated with currently released code. Jeffries et al. (2001) recommend using a separate integration machine, where the integration is conducted. XP literature emphasizes the importance of SCM automation to support XP practices (e.g. Jeffries et al. 2001; Bendix & Hedin 2002). In addition, it has been understood that an SCM tool can improve productivity. Jeffries et al. (2001) state that, in general, an SCM tool should be easy to use. Further they emphasize that there should be as few restrictions as possible in an SCM tool for example, no passwords, no group restrictions, as little ownership "hassle" as possible. Because of XP's collective ownership practice, an SCM tool should support concurrent development. According to the experiences of Lippert et al. (2002), optimistic concurrency control works a lot better than a locking mechanism in agile methods like XP.

As already mentioned, FDD requires that SCM be considered in FDD projects. According to Palmer & Felsing (2002), an SCM tool should be able to identify the source code for all the features that have been completed and able to maintain the history of changes to classes as they are enhanced. However, Palmer and Felsing (2002) also state that actual SCM needs are dependent on the nature and complexity of the software.

Pragmatic Programming emphasizes the importance of the SCM tool. Tip#23 (Hunt & Thomas 2000, p. 88) says, "always use source code control". Thus SCM tools should be used regardless of team size. According to Hunt & Thomas (2000), an SCM tool should at least have version control, but branching, baselining, build management and concurrent development are also seen as an important and helpful features. In addition, Pragmatic Programming stresses that everything should be under version control.

5. Discussion

In this section, the results of the analysis are discussed. Table 8 summarizes the results of the study and their implications. The discussion is divided into five subsections according to perspectives.

Table 8. Results and implications of the study.

Perspective	**Description of the results**	**Implications**
SCM approach	Only two of the methods take SCM explicitly into account. Most of the methods address SCM at some level via SCM tools.	SCM is a key component in the development of quality software and, therefore, it should not be neglected.
SCM planning	The literature of the agile methods does not mention anything about SCM planning.	Emphasis on SCM planning is needed, because it is a basis for successful SCM implementation.
Configuration identification	Only three of the methods emphasized that everything should be under version control.	With modern SCM tools "controlling everything" should be the tendency also in the other agile methods.
Change management	The agile methods understand that changes are inevitable in software development. However, change management approaches diverge.	Change management processes should be taken into account as a part of a project's SCM implementation.
SCM tools	Most methods emphasize the important role of SCM tools. The ability to revert to earlier versions of items was seen as the most important feature.	Methods should place more emphasis on comprehensive SCM solutions.

5.1 SCM approach

The results of this study revealed that only two of the analyzed agile methods take SCM explicitly into account (FDD and DSDM). Moreover, the literature of agile methods did not describe any SCM approach according to the traditional definition of SCM. Still, most of the methods take SCM into account via SCM tools. In the case of agile methods, SCM implementations do not have to be as heavy weight as in larger software projects. However, according to (Jonassen Hass 2002), effective software configuration management is crucial to the success of agile projects. The key point is that SCM is a main component in the development of quality software and it should not be neglected.

5.2 SCM planning

Agile methods literature does not mention SCM planning at all. One reason may be that agile methods value working software over comprehensive documentation (Beck et al. 2001). However, careful SCM planning is a basis for a project's SCM implementation (Abran & Moore 2001) and, therefore, it should not be neglected. It is typical for agile methods that documentation can be added later when there is time. However, this is not a recommended solution in the case of SCM plan. An SCM plan describes the guidelines for software configuration management, e.g., an identification scheme for configuration items. If there is no SCM plan, developers will use the SCM practices they consider best at the moment, producing a great variation among the SCM practices in use. Therefore, agile methods should place emphasis on SCM planning as well.

5.3 Configuration identification

Only three of the analyzed methods, FDD, DSDM and Pragmatic Programming, emphasize that every development artefact should be under version control. This means that artefacts are configuration items and need to be uniquely identified. In the case of XP, the literature emphasized that source code should be under version control. According to Leon (2000, p. 93), "modern SCM tools can efficiently and effortlessly handle any number of CIs without any problems."

Therefore, not only source codes but also every important development artefacts should be kept under version control.

Baselining strategies were not actually defined. Only DSDM suggested that baselining should be possible on a daily basis.

5.4 Change management

As noted above, changes are inevitable in software development. This has been accepted for a long time. Agile methods make no exception in this case. In fact, according to Cockburn & Highsmith (2001a), agility is about creating and responding to change. In fast-paced development there will be changes, and the important question is how they should be reacted. According to McConnell (1998), controlling changes is critical to a project's success. It protects the project from unnecessary change, ensures that all concerned parties are involved in decision-making and improves the quality of decisions made (McConnell 1998).

The results of this study show that different agile methods have quite different approaches towards change management. ASD suggests adapting to changes rather than controlling them. In FDD and DSDM, a change control process is part of the required SCM implementation. As Whitgift (1991) has stated, the level and formality of change control varies; large teams need strict and formal change control, but small teams can rely on less formal control. According to Boehm (2002), small teams and products are home ground for agile methods, which means that in agile projects the change management processes do not necessarily need to be as formal and rigid as in larger projects. Change management can also be a separate function regardless of the agile method. Thus, the main point is that change management should not be neglected.

5.5 SCM tools

The results of this study show that most of the agile methods highly appreciate SCM tools and the automation they can offer. According to Ronkainen & Abrahamsson (2003, p. 6), "pervasive use of version / configuration control is

one key ingredient in enabling fast-paced development work in an environment where seemingly harmless changes may cause bugs that are very difficult to locate and fix". This study supports their argument. Agile methods seem to relish having an ability to revert to earlier versions of development artefacts. Since rapid development and quick changes may lead to mistakes in development, it is important that earlier versions of development artefacts are accessible. According to Berczuk & Appleton (2003), version control is an essential part of making team software development effective, and agile methods are focused on teamwork (e.g. Abrahamsson et al. 2002). Therefore, version control is one of the most important features an SCM tool should have in an agile software development project. Other tool features that were seen as important were optimistic concurrency control, baselining and branching.

Overall, the literature on agile methods revealed that these methods have a very tool-focused viewpoint towards SCM. This tendency is like the one Leon (2000) has stated; the role played by SCM tools is becoming more and more important in today's software development environments. Leon (2000) has also stated that *increased business agility* is one of the SCM tool's benefits and means that "the company can be more agile and more responsive to the needs of the customer without compromising product quality." (Leon 2000, p. 189). SCM tools do not solve all configuration management problems, but they can be one step towards more effective software configuration management (Leon 2000). Therefore, agile methods should emphasise more comprehensive SCM solutions and understand the fact that SCM tools are only one part of a successful SCM implementation.

5.6 Summary

In this section, the results of the analysis were discussed in five subsections. The results of this study show that only two of the analyzed agile methods take SCM explicitly into account. Still, most of the methods take SCM into account at some level via SCM tools. Because software configuration management is crucial to the success of agile development, agile methods should emphasise more comprehensive SCM solutions.

6. Conclusions

Agile software development methods have gained significant attention in the software engineering community in the last few years. They place more emphasis on individuals, interactions, working software, customer collaboration, and responding to change, rather than on processes, tools, documentation, contracts and plans. SCM is a method of bringing control to the software development process and is known as an inseparable part of quality-oriented product development regardless of development method. On the other hand, software configuration management is often considered bureaucratic method, an approach that causes additional work and more documentation. However, the value of SCM should not be underestimated in the case of agile software development methods. Currently, there are very few studies of software configuration management in agile methods.

The aim of this publication was to systematically review the existing literature on agile software development methodologies from an SCM point of view. Therefore, analytical SCM lenses based on existing literature were constructed, including the following five perspectives:

SCM approach: This perspective analyzed how software configuration management is addressed in a method and explored each method's SCM-related practices. It was found that only two of the methods take SCM explicitly into account. However, most of the methods address SCM at some level via SCM tools.

SCM planning: The purpose of this perspective was to find out what the agile methods define as SCM planning. The results show that agile methods literature does not mention SCM planning at all. In software configuration management literature, SCM planning has been described as a basis for successful SCM implementation. Therefore, agile methods should also place emphasis on SCM planning.

Configuration identification: In this perspective, it was analyzed what the agile methods defined as item identification and baselining. The results of the study show that only three of the existing agile methods stipulate every development artefact should be under version control. However, modern SCM tools can

handle any number of configuration items without any problems and, therefore, "controlling everything" should be the tendency in the other agile methods as well. Only DSDM has addressed baselining by suggesting that it should be possible to carry out baselining daily.

Change management: The purpose of this perspective was to clarify what agile methods define as change management. It was found that different agile methods take quite different approaches. For example, ASD suggests adapting to changes rather than controlling them. Respectively, in FDD and DSDM, the change controlling process is part of the required SCM implementation. Thus, change management processes should be defined and taken into account as part of a project's SCM implementation

SCM tools: This perspective analyzed the role of software configuration management tools in agile methods. The results show that most of the methods emphasize the importance of SCM tools, and the ability to revert earlier versions of items is seen as the most important feature. However, while tools are an important part of software configuration management, agile methods should stress more comprehensive SCM solutions.

In conclusion, the results of this study showed that there are lacks in the SCM approaches in the different agile methods. The positive result was that software configuration management had not been completely forgotten. However, it is clear that SCM should be a key ingredient in fast-paced agile development.

References

Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J. 2002. Agile Software Development Methods: Review and Analysis. Espoo, VTT Electronics, 107 p. VTT Publications 478.
http://www.vtt.fi/inf/pdf/publications/2002/P478.pdf.

Abrahamsson, P., Warsta, J., Siponen, M. T., Ronkainen, J. 2003. New Directions on Agile Methods: A Comparative Analysis. Proceedings of the 25th International Conference on Software Engineering (ICSE'03).

Abran, A., Moore, J. 2001. SweBok: Guide to the Software Engineering Body of Knowledge, Trial Version 1.0, California: IEEE Computer Society Press.

Ambler, S. W. 2002a. Agile Modeling: Best Practices for the Unified Process and Extreme Programming. New York: John Wiley & Sons.

Ambler, S. W. 2002b. Introduction to Agile Modeling. A Ronin International, Inc. White Paper.

Asklund, U., Bendix, L. 2002. A Study of Configuration Management in Open Source Software Projects. Software, IEE Proceedings, Vol. 149, No. 1, pp. 40–46.

Baskerville, R., Levine, L., Pries-Heje, J., Ramesh, B., Slaughter, S. 2001. How Internet Companies Negotiate Quality. IEEE Computer, Vol. 34, pp. 51–57.

Beck, K. 1999a. Embracing Change with Extreme Programming. Computer, Vol. 32, No. 10, pp. 70–77.

Beck, K. 1999b. Extreme Programming Explained: Embrace Change. Reading, Massachusetts: Addison-Wesley.

Beck, K., Beedle, M., Bennekum, A. V., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J. Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D. 2001. Manifesto for Agile Software Development [Web-document]. Available: http://agilemanifesto.org/. [Referenced 14.5.2003].

Bendix, L., Hedin, G. 2002. Summary of the subworkshop on extreme programming. Nordic Journal of Computing, Vol. 9, No. 3, pp. 261–265.

Berczuk, S. P., Appleton, B. 2003. Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Boston: Addison-Wesley.

Boehm, B. 2002. Get Ready for Agile Methods, with Care. Computer, Vol. 35, No. 1, pp. 64–69.

Buckley, F. J. 1996. Implementing Configuration Management: Hardware, Software and Firmware, 2nd ed. New York: IEEE Press.

Charette, R. 2001. The Decision Is In: Agile Versus Heavy Methodologies. Cutter Consortium e-Project Management Advisory Service, Vol. 2, No. 19.

Coad, P., LeFebvre, E., De Luca, J. 2000. Java Modeling in Color with UML: Enterprise Components and Process. Prentice Hall.

Cockburn, A., Highsmith, J. 2001a. Agile Software Development: The Business of Innovation. Computer, Vol. 34, No. 9, pp. 120–122.

Cockburn, A., Highsmith, J. 2001b. Agile Software Development: The People Factor. Computer, Vol. 34, No. 11, pp. 131–133.

Cockburn, A. 2002. Agile Software Development. Boston: Addison-Wesley.

Compton, S. P., Conner, G. R. 1994. Configuration Management for Software. New York: Thomson Publishing Company.

Conradi, R., Westfechtel, B. 1996. Version Models for Software Configuration Management. ACM Computing Surveys, Vol. 30, No. 2, pp. 232–282.

Conradi, R., Westfechtel, B. 1998. Software Configuration Management and Engineering Data Management: Differences and Similarities. Proceedings 8th International Workshop on System Configuration Management. Brüssel. LNCS 1439. Springer-Verlag. Pp. 95–106.

Conradi, R., Westfechtel, B. 1999. SCM: Status and Future Challenges. Ninth International Workshop on Software Configuration Management (SCM-9).

Crispin, L., House, T. 2003. Testing Extreme Programming. Boston, Addison-Wesley.

Crnkovic, I., Persson Dahlqvist, A., Svensson, D. 2001. Complex Systems Development Requirements – PDM and SCM Integration.

Crnkovic, I., Asklund, U., Persson Dahlqvist, A. 2003. Implementing and Integrating Product Data Management and Software Configuration Management. Artech House.

Dart. S. 1990. Spectrum of Functionality in Configuration Management Systems. Technical Report CMU/SEI-90-TR-11.

Dart, S. 1994. Concepts in Configuration Management Systems. Technical Report. Software Engineering Institute, Carnegie Mellon University, Pittsburgh.

Estublier, J. 2000. Software Configuration Management: A Roadmap. The Future of Software Engineering, 22nd International Conference on Software Engineering (ICSE 2000).

Feiler, P. H. 1990. Software Configuration Management: Advances in Software Development Environments. Technical Report. Software Engineering Institute, Carnegie Mellon University, Pittsburgh.

Highsmith, J. A. 2000. Adaptive Software Development: A Collaborative Approach to Managing Complex Systems. New York, NY, Dorset House Publishing.

Highsmith, J. 2002a. Agile Software Development Ecosystems. Boston, Addison-Wesley.

Highsmith, J. 2002b. What is Agile Software Development. Crosstalk, the Journal of Defense Software Engineering, October.

Hunt, A., Thomas, D. 2000. The Pragmatic Programmer. Addison Wesley.

IEEE 1042-1987. IEEE Guide to Software Configuration Management. IEEE Standards Collection – Software Engineering, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

IEEE 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology. IEEE Standards Collection – Software Engineering, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

IEEE 828-1990. IEEE Standard for Software Configuration Management Plans. IEEE Standards Collection – Software Engineering, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

IEEE 828-1998. IEEE Standard for Software Configuration Management Plans. IEEE Standards Collection – Software Engineering, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Jeffries, R., Anderson, A., Hendrickson, C. 2001. Extreme Programming Installed. NJ: Addison-Wesley.

Jonassen Hass, A. M. 2002. Configuration Management Principles and Practices. Addison-Wesley.

Lehman, M. 1980. Programs, lifecycles and laws of software evolution. Proceedings of IEEE. Vol. 68, No. 9.

Leon, A. 2000. A Guide to Software Configuration Management. Boston: Artech House.

Lippert, M., Roock, S., Wolf, H. 2002. Extreme Programming in Action: Practical Experiences from Real World Projects. England: John Wiley & Sons Ltd.

Martin, R. C., Hall, P. 2001. Agile processes. Chapter from "Agile Development: Principles, Patterns, and Process".

Martin, R. C. 2002. Agile Software Development: Principles, Patterns, and Practices. Upper Saddle River, NJ, Prentice-Hall.

Maurer, F., Martel, S. 2002. Extreme Programming: Rapid Development for Web-Based Applications. IEEE Internet computing, Vol. 6, Issue 1, pp. 86–90.

McConnell, S. 1998. Software Project Survival Guide. Redmond, Washington, Microsoft Press.

Palmer, S. R., Felsing, J. M. 2002. A Practical Guide to Feature-Driven Development. Upper Saddle River, NJ, Prentice-Hall.

Paulk, M. C. 2001. Extreme Programming from a CMM Perspective. Paper of XP Universe.

Persson Dahlqvist, A., Asklund, U., Crnkovic, I., Hedin, A., Larsson, M., Ranby, J., Svensson, D. 2001. Product Data Management and Software Configuration Management: Similarities and Differences. The Association of Swedish Engineering Industries.

Poppendieck, M., Poppendieck, T. 2003. Lean Software Development – An Agile Toolkit. Upper Saddle River, NJ: Addison Wesley.

Pressman, R. S. 1997. Software Engineering: a Practitioner's approach, 4th ed. New York: McGraw-Hill Companies.

Rahikkala, T. 2000. Towards Virtual Software Configuration Management. A Case Study. Espoo, VTT Electronics, 110 p. + app. 57 p. VTT Publications 409. http://www.vtt.fi/inf/pdf/publications/2000/P409.pdf.

Ronkainen, J., Abrahamsson, P. 2003. Software Development Under Stringent Hardware Constraints: Do Agile Methods Have a Chance? 4th International Conference on Extreme Programming (XP 2003).

Stapleton, J. 1997. Dynamic Systems Development Method – The Method in Practice. Addison Wesley.

Schwaber, K., Beedle, M. 2002. Agile Software Development With Scrum. Upper Saddle River, NJ, Prentice-Hall.

Taramaa, J. 1998. Practical Development of Software Configuration Management for Embedded System. Espoo, VTT, 147 p. + app. 110 p. VTT Publications 366. http://www.vtt.fi/inf/pdf/publications/1998/P366.pdf.

Tichy, W. 1988. Software Configuration Management Overview.

Weatherall, B. 1997. A Day in the Life of a PVCS Road Warrior: Want to Get PVCS Organized Quickly in a Mixed-Platform Environment? Technical paper, Synergex International Corporation [Web-document].
Available: http://www.pvcs.synergex.com/oll_files/synergex-151.asp.
[Referenced 14.5.2003].

Whitgift, D. 1991. Methods and Tools for Software Configuration Management. England: John Wiley & Sons Ltd.

Zeller, A. 1996. Software Configuration with Feature Logic. Proceedings of the Workshop on Knowledge Representation and Configuration Problems.

Published by

Series title, number and report code of publication

VTT Publications 514
VTT–PUBS–514

Author(s)
Koskela, Juha

Title

Software configuration management in agile methods

Abstract

The development of good quality software is a critical element of successful competition for today's software market shares. However, software products are becoming larger and more complex; therefore, the development of quality software is neither easy nor rapid. Agile software development methods focus on generating early releases of working products. They aim to deliver business value immediately from the beginning of the project. Regardless of the development method in use, it is important that software development be under control. Software configuration management (SCM) is known as a method of bringing control to the software development process, and thus, proper application of SCM is a key component in the development of quality software. However, currently very few studies on software configuration management in agile methods exist; hence this study.

The aim of this publication is to systematically review the existing literature on agile software development methodologies from an SCM point of view. First, analytical SCM lenses based on existing SCM literature are constructed. Second, existing agile methods are analyzed using the lenses constructed. The results show that only two of the existing agile methods take SCM explicitly into account, but most of the methods highly value SCM tool support and its ability to revert to earlier versions of development artefacts. Nonetheless, the basis for successful SCM implementation, SCM planning, has been completely forgotten.

Keywords
software configuration management (SCM), agile methods, extreme programming, software development methods

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

ISBN	Project number
951–38–6259–3 (soft back ed.) 951–38–6260–7 (URL:http://www.vtt.fi/inf/pdf/)	E2SU00055

Date	Language	Pages	Price
December 2003	English	54 p.	B

Name of project	Commissioned by
Agile software technologies (UUTE2)	VTT

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

VTT PUBLICATIONS

Tätä julkaisua myy	Denna publikation säljs av	This publication is available from
VTT TIETOPALVELU PL 2000 02044 VTT Puh. (09) 456 4404 Faksi (09) 456 4374	VTT INFORMATIONSTJÄNST PB 2000 02044 VTT Tel. (09) 456 4404 Fax (09) 456 4374	VTT INFORMATION SERVICE P.O.Box 2000 FIN–02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374

ISBN 951–38–6259–3 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–6260–7 (URL: http://www.vtt.fi/inf/pdf/)
ISSN 1455–0849 (URL: http://www.vtt.fi/inf/pdf/)