Software Configuration Management Practices for eXtreme Programming Teams

Ulf Asklund, Lars Bendix, Torbjörn Ekman
{ulf | bendix | torbjorn}@cs.lth.se
Department of Computer Science
Lund Institute of Technology
Sweden

Abstract

Extreme Programming (XP) is becoming popular as a software development method and there is quite a lot of literature describing its philosophy and practices. However, in all of this literature Software Configuration Management (SCM) is almost never mentioned explicitly, leaving XP practitioners with the impression that SCM is not needed and SCM people with the impression that XP is not sound from an SCM perspective.
We carried out a more profound analysis of XP and its practices seen from an SCM perspective. We found that in general XP and its practices do not go against common SCM standards, if we take into consideration that the XP context is different from that of more traditional projects. However, some SCM aspects need to be made explicit and a number of SCM-specific sub-practices need to be added to make XP a complete and sound development method seen from an SCM perspective. We report on how we implemented "our findings" on several dozen XP projects and our experience from doing this through several iterations.

1. Introduction

Ever since Kent Beck published his seminal paper [Beck99] on eXtreme Programming (XP) a lot has been written about XP and its philosophy and practices, and the use of XP as a development method is spreading. Some years ago our department decided to use XP on second year student projects [HBM03], and we realised that although the XP literature, like [Beck00], [JAH01] and [chromatic03] says a lot about XP practices and explains how they support each other, it hardly mentions Software Configuration Management (SCM) and how

to do that on XP projects. It says Frequent Releases, but how can we do that according to SCM rules and still keep up the speed in a project? It says that we should have Collective Code Ownership, but that sounds frightening to many seasoned SCM people – can it be done, and if so, how should it be done? Those questions, and many others regarding SCM and XP, are left unanswered – or at best hinted at implicitly – in the current literature.

So we set out to analyse XP and SCM and their interactions to see whether XP is actually a sound and complete development method seen from an SCM perspective. We suspected that it would be the case, as most reported XP projects seem to be successful. However, we claim that no development project can be a success without the use of SCM, and as SCM is not explicitly mentioned in XP and its practices, this cannot be the full story. The success of XP projects must derive from doing something more than the basic XP practices. Implicit SCM processes must have been followed on these projects. Our aim was to bring out these processes such that they could be stated explicitly and our students could have this additional support for their projects. Because they are not seasoned developers, they have no previous implicit or intuitive experience to help them.

We expect other XP projects to be able to benefit from our findings – and we expect SCM people to do so too. SCM is a generic service and one size does not fit all, so each specific (type of) project has to tailor the way SCM is carried out to its specific needs – and so has also XP projects.

We start by a brief presentation of SCM and those practices of XP that we consider relevant from an SCM perspective. Then we go on to a careful, in-depth analysis of each of the relevant XP practices with respect to how sound and complete they are seen from an SCM perspective. This uncovers a number of weaknesses where SCM requirements are not met. In the second part of the analysis, we cover all the SCM activities needed and analyse to which extent they are covered by the XP practices. Again some weaknesses are revealed. Subsequently we design a number of SCM-related sub-practices that will satisfy those SCM requirements that were not met by the standard XP practices to complete the development method from an SCM perspective. We then briefly tell the story of how we have implemented some of these new SCM sub-practices on a number of XP projects and our experience from several iterations of improvement. Finally, we draw our conclusions and indicate what we think could be done in the next iteration.

2. Background

This chapter will provide the reader with a brief overview of important SCM activities and XP practices relevant to SCM. Subsequent chapters will use the structure, terminology and concepts introduced here.

2.1 Software Configuration Management

SCM is traditionally considered as consisting of the four activities: Configuration Identification, Configuration Control, Configuration Status Accounting and Configuration Audit [Berlack92], [Leon00]. However, these activities reflect mostly the part of a development project that has relations to the customer. In the context of analysing XP practices, we have chosen to include also developer oriented aspects of SCM such as: version control, build management, workspace management, concurrency control, change management and release management [Babich86], [MP97] and [AB01].

Configuration Identification:

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

Configuration Control:

Activities comprising the control of changes to a configuration item after formal establishment of its configuration documents. Control includes evaluation, co-ordination, approval or disapproval, and implementation of changes. Implementation of changes includes engineering changes and deviations, and waivers with impact on the configuration.

Configuration Status Accounting:

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

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

Configuration Audit:

Examination to determine whether a configuration item conforms to its configuration documents. Functional configuration audit: a formal evaluation to verify that a configuration item has achieved the performance characteristics and functions defined in its configuration document. Physical configuration audit: a formal evaluation to verify the conformity between the actual produced configuration item and the configuration according to the configuration documents.

Version Control:

The possibility to store, recreate and register the historical development of an item (document or source code) is a fundamental characteristic of a version control system. The tool has to minimize the storage space needed to keep all versions of an item. It has to impose a structure on how versions can develop

from each other. And, finally, it has to keep track of relevant information about the different versions.

Build Management:

Build management handles the problem of putting together and compile modules in order to create a running system. The description of dependencies and information about how to compile items is given in a system model, which is used to derive object code and to link it together. The build process can be automated and implement as well minimal builds as compilation in heterogeneous environments and support for parallel compilation.

Workspace Management:

The different versions of the documents in a project are kept in a repository by the version control tool. Because these versions have to be immutable, developers cannot be allowed to work directly within this repository. They have to take out a copy of the document, modify it, and add the modified copy to the repository. Workspace management must provide functionality to create a workspace from a selected set of files from the repository. When the developer has finished carrying out modifications, he needs to add the changed documents and modules to the repository. Working on the changes, a developer would need to be able to update his workspace with (selected) additions from the repository.

Concurrency Control:

If we want to allow several developers to work on the system at the same time, we must also provide mechanisms to synchronize their work. The problem that can occur is that more than one developer in his workspace makes a change to the same document or module. If this situation is not detected - or avoided - the second developer will overwrite the first developer's change when he adds his workspace to the repository.

Change Management:

There are multiple and complex reasons for changes and change management covers all types of changes to a system. It includes tools and processes, which support the organization and tracking of changes from the origin of the change to the approval of the actually implemented source code. Various tools are used to collect data during the process of handling a change request. It is important to keep traceability between the change request and the actual implementation - in both directions. Change management data can also be used to provide valuable metrics about the progress of project execution.

Release Management:

Release management deals with both the formal aspects of releasing to the customer and the more informal aspects of releasing to the project. For a customer release we need to carry out a configuration audit before the actual release. This includes verifying that the produced package (application, documentation, help files, etc) is actually installable. Releasing changes to the

project is a matter of how to integrate changes from developers. We need to decide on when and how that is done – and in particular on the "quality" of changes before they can be released. In order to re-create a release we use a bill-of-material (BOM) that records *what* went into a product and *how* it was built.

2.2 eXtreme Programming

Not all of XP's practices are directly related to an analysis from an SCM perspective. We have identified the following as the most directly related: Collective Code Ownership, Continuous Integration, Frequent Releases, Refactoring and Planning Game.

Collective Code Ownership:

This practice means that everyone owns all parts of the code and consequently is free to change it at will. There is no notion of assigning ownership of code to individual developers or pairs and having them be responsible for that code. If a pair working on a particular story feels the need to change some other code to make their own story work, they are free to do so at any time without having to ask and wait for someone else to do it for them. In fact they are encouraged to add, change or refactor code whenever they see that it can be improved. If they need additional information to carry out a change, they can look up the pair that made the code in the first place and discuss the change with them. As a consequence of practising Collective Code Ownership there will inevitably be parallel work, that is, two or more pairs changing the same code at the same time.

Continuous Integration:

This means that changes have to be integrated into the common production code immediately when a task or story is done. As a part of the integration process you first integrate the latest production code into your own changes, then you must run all the unit tests once again to verify that they still work and that your new code (or changes done by others) does not break the system. Only then you can "release" your code and integrate it into the pool of common production code. Continuous Integration also means that a pair should continuously integrate the latest version of the common production code into their workspace.

Frequent Releases:

This practice means that an XP project should try to put the system into production use as early as possible and even before it has solved the whole problem. It should also subsequently do new releases very frequently, which can be anywhere from a daily to a monthly basis. What has to be released is not a demo that is commented on and then set aside, but (part of) an actual system that the customer puts into production use. The team and the customer can use Frequent Releases as an aid in steering the project and it also gives the team and the customer confidence in the project to see that something useful is actually produced and delivered.

Refactoring:
This practice is probably the trickiest one to use. The idea is that XP is very code driven. After the initial zero-th version where the general architecture and design is laid out, everything is focused on implementing stories and making them work. After a task or story has been completed, but before it is released to the common repository, the code is taken through a series of transformations to simplify both the code and the design. Refactoring should be carried out as a series of steps that are reversible, so you can always back out if a refactoring does not work. For each step in the refactoring process you make sure that the transformed code does not break any of the unit tests.

Planning Game:
This practice handles scheduling of an XP project. At the start of each iteration, it is decided what should go into it. XP stresses that it is a "game" between the customer, who provides resources (time and money) and requirements (stories that the customer has written beforehand), and the developers, who estimate the costs and risks of implementing each requirement. The "game" then consists in the customer making an informed choice of which requirements should actually be implemented in the following iteration and which should be postponed to subsequent iterations or even abandoned. If there is not a perfect match between resources and estimate, the customer has the option to either remove some requirements from this iteration or to allocate more resources. He could also ask the developers to reconsider the estimate in case it "surprises" him, or the developers could prompt the customer for more information needed to do the estimate. However, it is important that each player sticks to his role.

3. Analysis

In this chapter, we will do a more careful in-depth analysis of each of these XP practices with respect to how sound and complete they are seen from an SCM perspective. This gives rise to the introduction of a number of SCM-related sub-practices. We also cover the general SCM activities needed in any type of project and analyse to which extent they are covered by the XP practices. In many cases, XP already takes care of the SCM requirements, but in some cases our more explicit SCM focus results in a number of SCM-related sub-practices that are needed to make the XP development process complete. Whenever the analysis ends with the finding that a sub-practice is needed, there is a reference within curly brackets to the definition and description in chapter 4.

3.1 From XP practices

We start with the XP practices generally described above and for each practice we analyse if, and if so how, this practice either implements SCM requirements or makes it harder to maintain a proper SCM level.

Collective Code Ownership:

A direct consequence of Collective Code Ownership is that the copy-merge work model is the only realistic way to synchronize developers {Use copy-merge work model}, see also workspace management below.

Continuous Integration:

The drawback of the copy-merge model and optimistic check-out is the potential risk of merge conflicts that can be hard to resolve. Continuous Integration minimizes this drawback. An important clarification is that integration means both updating the workspace and committing to the main development line.

This practice may be tailored a bit if branches are used. If, for example, each story is implemented on its own branch in order to increase traceability, other changes should not be mixed into the branch during the development.

Refactoring:

A refactoring differs from a "normal" implementation of a story (functionality) in that it often is more global. The actual changes made due to the refactoring is scattered over a larger part of the code compared to a "normal" implementation, which is more local. A consequence of this is that the risk for merge conflicts increases when refactorings are made in parallel with other changes (or indeed other refactorings).

To reduce this drawback with refactorings the developer should really try to divide the refactoring into smaller steps, and for each step integrate, test, and commit {Incremental refactoring}. In this way possible merge conflicts are found as soon as possible and the size of each conflict is smaller.

Applying the copy-merge model {Use copy-merge work model}, as proposed in workspace management below, also means that it is possible to continuously maintain a "virtual split-combine", i.e. not a long-lived static split, but a split continuously changing with the stories/tasks currently implemented. To reduce merge conflicts hard to resolve, an impact analysis for each story evaluating possible conflicts with other potential parallel stories should be made {Impact analyse refactorings}. This is especially important for large refactorings, as these tend to create merge conflicts.

Planning Game:

As part of the Planning Game, each story is prioritised and the cost to implement it is estimated. In addition to the cost to implement each story by itself, there will be a cost to integrate them and to resolve possible merge conflicts. If the costs due to coordination and integration get too high it may have been better to change the order in which they were implemented (and change the stories implemented concurrently). Thus, including an impact analysis as part of the Planning Game, predicting possible large integration problems and take these into consideration when establishing the plan for the next iteration, can reduce

the total cost of implementing a certain set of stories {Impact analysis of stories as part of Planning Game}.

Frequent Releases:
A release will always require work to a certain extent. According to SCM there are some basic requirements that should be fulfilled: physical and functional audit, possibility to recreate a release (its BOM) and to bug fix an old release without disturbing the current development. In addition the possibility to "story freeze" and to fix the final bugs before a release in parallel with implementing new functionality to later releases is often desirable (even though XP tries to make such time period as short as possible). The question is if all this is really needed in XP and how can we reduce the release cost to a minimum and automate as much as possible.

XP takes care of some of the requirements by some of the other practices: Continuous Integration makes sure there are no large integrations that has to be done before we release, Planning Game makes us certain of what stories should be included in the next release, test-first makes sure we always have both unit tests and acceptance test ready to run as functional audit. What remains is the physical audit, i.e. to make sure we deliver all the files and documents and the correct version of them {Physical audit in release process}. To fit into the XP philosophy the cost of this audit should be minimized by automating it, e.g. using scripts and build tools supporting BOM-reports.

3.2 From SCM activities

In this part of the analysis we cover the SCM activities described in chapter 2 and analyse to which extent they are covered by the XP practices. When they are not automatically covered by the core XP practices, we refer to SCM sub-practices defined in chapter 4.

Configuration Identification:
In XP there is no practice about version control and thus no proposal of what items should be configuration items. However, in practice most XP teams use version control to synchronize concurrent development and to be able to manage Collective Code Ownership. Applying version control requires the selection of configuration items and a decision of how they should be structured {Define configuration items and their structure}. To avoid unnecessary work, this should be done early in the project. Many version control tools do not support rename and move of configuration items very well, and a lot of restructuring will unfortunately be time consuming and may even result in lost history log for the moved items.

What items should be configuration items? Source code, unit tests, acceptance tests, and stories are all items that may change and that are important and should

be stored in a repository, i.e. configuration items. Spike code, class-files, and private stuff should probably not.

Configuration Control:

Most of the requirements part of configuration control is taken care of by XP. Change requests (called stories in XP) are discussed, prioritized, planned, and decided upon during the Planning Game, which is the XP equivalent to the formal CCB (change control board) in SCM.

However, the demand to be able to trace the actual source code changes to the story they implement (and vice versa) is not automatically covered. If a version control tool is used, it is easy (at almost no extra cost) to include the story number in the commit comment, making it possible to see why (which story) a certain version was created {Trace changes to stories}. It is harder to implement traceability from a certain story to the actual changes made to implement the story. This requires both a management of stories that allows recording this extra information, and it puts the demand on the developers to actually do this registration when (or after) implementing the story.

Configuration Status Accounting:

Most of the status accounting activity is taken care of by the tracker keeping track of all stories and tasks and their current state, which is information needed by the coach and customer to follow the progress of the project. Also the developers use this information to synchronize concurrent development.

Configuration Audit:

Functional audit can be automated thanks to test first. When the new functionality is implemented we know that all the tests are already written, i.e. there is no risk of not being able to do a functional test. However, XP says nothing about practices for a physical audit, which is required for each release according to SCM. To be able to release frequently the physical audit should be automated, e.g. using build tools or scripts generating the BOM {Physical audit in release process}.

Version Control:

Use a version control tool {Use a version control tool}. Awareness of what have happened with the configuration items, integration (including merge), the long transaction model preventing a developer to commit not integrated code into the common repository, and branches are examples of indispensable support given by a version control tool.

Build Management:

Sufficient tool support should be used to build both the source code and to generate documentation. Some of this support is included in the compiler, but if not, other dedicated tools should be used {Use a build tool}.

Workspace Management:
This is an important SCM functionality in an XP project making it possible to allow Collective Code Ownership. Workspaces implement the copy-merge model {Use copy-merge work model}, which makes it possible to work in isolation with the entire product (or as much of it as needed), integrate with other changes, and perform all tests before committing it to the common repository (project release). It thus makes it possible to keep the repository clean from non functional code according to unit tests, which is important using Collective Code Ownership {Keep the repository clean}.

Concurrency Control:
XP does not explicitly force any concurrency control, but the only realistic choice is the copy-merge model, see workspace management above. Turn-taking and split-combine do not work together with Collective Code Ownership.

Change Management:
Configuration control, {Trace changes to stories}, and configuration status accounting increase the group awareness of what changes are currently implemented and by whom, and it also makes it easier to trace what has been done earlier. However, the developers themselves can further increase the group awareness and help fellow developers to later understand why certain changes were made to the code by writing proper commit comments {Write proper commit comments}. A comment should normally include: what change you made and why, e.g. what refactoring, or which task you implemented. Do not write details of how you made the change, it should be "obvious" from reading the code. Using a version tool {Use version control} makes it possible to compare versions and view the diff, i.e. what was changed.

Release Management:
In XP it should be possible to release frequently. Continuous Integration ensures that the latest development is always integrated and committed to the common repository and test-first makes it possible to always do a functional audit. Still, support for physical audit remains, see configuration audit above {Physical audit in release process}. To be able to release with short notice and with little effort the release process, including the packaging of all included items, should be automated and optimized (slim) {Automate and optimise the release process}.

4. Design

In this chapter, we define the SCM sub-practices we found needed during the analysis above and put them into the XP context. For each sub-practice we analyse what other sub-practices and XP practices it supports and what are required to make it work.

Incremental refactoring:

Definition: Break down large refactorings into smaller steps (increments) and commit each increment directly when integrated and tested.
Motivation: The developer making the refactoring can better control the change and more easily reverse mistakes. It utilizes Continuous Integration and minimizes the risk of complex merge conflicts between refactoring and other committed changes. For all the other developers, each commit makes it possible for them to integrate into their workspaces and thus avoid a large merge at the end of the entire large refactoring.
Supports: Refactoring (since we reduce the drawback of large refactorings), Continuous Integration (since each increment can be integrated).
Requires: Refactoring generally requires Collective Code Ownership and Coding Standards.

Impact analyse refactorings:

Definition: An impact analysis of possible (merge) conflicts with other stories in the current iteration should be done for large refactorings (written as stories).
Motivation: Reduces the number of hard merge conflicts due to refactorings. It also increases the developers' awareness of planned and ongoing large refactorings.
Supports: {Impact analyse stories as part of Planning Game}, Planning Game, {Use copy-merge model}, Collective Code Ownership.
Requires:

Use copy-merge work model:

Definition: Use the copy-merge work model, i.e. all developers have their own private workspace with the entire (necessary) code base in which they can implement tasks, integrate, and test before committing their changes back to the common repository. Use long-transaction model.
Motivation: To give tool support for the commit and automatically verify that all the tested code can be committed without any merge has to be done in the common repository the long-transaction model can be used.
Supports: {Keep the repository clean}, Refactoring, Collective Code Ownership.
Requires: {Use a version control tool}.

Impact analyse stories as part of Planning Game:

Definition: Do an impact analysis of the planned stories and include the cost of possible merge conflicts in cost estimate for each story.
Motivation: Results in a more precise cost prediction of stories as input to the Planning Game, and gives knowledge of how the total cost can be reduced by moving stories between iterations to avoid merge conflicts.
Supports: Collective Code Ownership, {Use copy-merge work model}.
Requires: Planning Game, {Impact analyse refactorings}.

Physical audit in release process:

Definition: Include a physical audit in the release process, both for demo and production releases. Should be automated to cope with “Frequent Releases”.
Motivation: To avoid small mistakes in the release process (e.g. to forget to include a configuration file in the release package) that may make the release unusable. Reduces the efforts needed to release.
Supports: Frequent Releases (if automated).
Requires:

Define configuration items and their structure:

Definition: Choose which items should be configuration items and create a structure for them. Should be done early in the project.
Motivation: Most version control tools do not support move or rename of stored items. To early create a structure reduces the need of these operations. (Note this is a major drawback of most tools especially for XP projects where changes should be embraced not hindered.)
Supports: {Use a version control tool}, {Automate and optimise the release process}.
Requires: {Use a version control tool}.

Trace changes to stories:

Definition: Each change set, typically each commit, should be traceable to the story and task it implements.
Motivation: It further emphasizes the by XP proposed work model to implement only one specific task at a time.
Supports: Continuous Integration, Collective Code Ownership.
Requires: {Use version control tool including functionality to manage meta-data, e.g. commit comments}.

Write proper commit comments:

Definition: Write commit comments describing why you made the changes and, at a high level, what changes you made. Do not write detailed descriptions of the changes, they should be understood reading the code alone.
Motivation: To increase the awareness between fellow developers, and to make it easier to understand and make changes to code written by other developers.
Supports: Collective Code Ownership, {Use copy-merge work model}.
Requires: {Use a version control tool supporting commit comments}.

Automate and optimise the release process:

Definition: Make it possible to automatically create a release, i.e. to package and send (or burn) the correct version of all needed files and documents.

Motivation: To release often and to be able to release on short notice is prioritized in XP and should thus be supported as much as possible, decreasing the cost for a single release.
Supports: Frequent Releases, on-site customer.
Requires: {Physical audit in release process}.

Use a version control tool:

Definition: Use a version control tool to store the common configuration items. Let all developers synchronize with this common repository.
Motivation: It is important that all developers trust the repository and dare make changes to the code. Version control both helps to synchronize concurrent changes and makes it possible to undo mistakes. It also enables different strategies of how to parallelize development, management of "story-freeze" close to release, and maintenance of old releases.
Supports: {Use copy-merge work model}, Continuous Integration, Frequent Releases, Collective Code Ownership, {Trace changes to stories}, {Write proper commit comments}.
Requires:

Use a build tool:

Definition: Use a build tool to build the executing product from the source code, but also to generate documentation (API, e.g. using java doc).
Motivation: To integrate often and to always test the product implies that it has to be built often. A build tool makes this less demanding. Also generating documentation reduces the cost of a release.
Supports: Continuous Integration, Frequent Releases.
Requires: Coding Standard.

Keep the repository clean:

Definition: The code checked in to the common repository should always work, i.e. it must always have been compiled and all unit tests should run without errors.
Motivation: All developers continuously integrate their workspace with the repository, i.e. an error in the repository will very fast propagate to all the developers destroying their build.
Supports: Frequent Releases, Collective Code Ownership, Continuous Integration, {Use copy-merge work model}.
Requires: {Use a version control tool}, Continuous Integration, Testing.

5. Implementation and experience

For the past three years, our department has used XP in a team-programming course for undergraduate students to introduce software engineering. The course is divided in a theory part to teach the XP methodology and a project part where the students practice team development using XP.

The theory part consists of seven lectures and four lab sessions. The lectures cover the 12 standard XP practices and the lab sessions introduce techniques and tools to support these practices.

The project part of the course is carried out in teams of 8-12 developers. Each team also has two coaches who guide the team members through the development process. The project is divided in 6 iterations, each consisting of 2 hours of team planning, 4 hours individual spike-time, and 8 hours team program development. Spike-time is used to experiment and study relevant topics for the user-stories currently being developed. During the 8 hours team development the entire team is located in one room, implementing customer supported stories while practising the techniques taught in the theory part of the course. For more details, please see [HBM03].

Many things have to be taught to the students, so the time available for SCM is limited to one 2-hour lecture and one 2-hour computer lab. The lecture is placed early in the course as the first in-depth study of selected XP practices, preceded only by a brief introduction to the 12 XP practices and software engineering in general. The computer lab that follows the lecture increases the students' understanding of the SCM-related practices and introduces tool support for basic SCM using CVS [Berliner90]. Two more lab exercises, supporting other lectures, use the CVS tool but focus on tools, techniques and practices for Test First and Refactoring respectively. Even though these labs are not strictly SCM-related, they do, however, increase the students' skills in using CVS.

So after the *course part*, all students were aware of – and had used – the following SCM-related sub-practices: {Use a build tool}, {Use a version control tool}, {Use copy-merge work model}, {Keep the repository clean} and {Incremental refactoring}. During the *project part* everyone used the first three sub-practices right from the start. Quite a few groups were reluctant to adopt the sub-practices {Keep the repository clean} and {Incremental refactoring}, and only did that after experiencing the problems of not doing so.

There was no formal SCM education at all during the project time apart from what the teams decided to do on their own. Most teams used some of the spike time to gain further knowledge of SCM (primarily the CVS tool). This sometimes resulted in tutorials or checklists used by all team members. Through

the coaches they also had access to – and sometimes also help in implementing – the rest of the SCM-related sub-practices.

The following SCM-related sub-practices were used by one or more individual groups: {Automate and optimise the release process}, {Physical audit in release process} and {Write proper commit comments}. Groups that used these sub-practices found them very helpful and with a high return-on-investment – especially for the first two sub-practices. Groups that did not implement these sub-practices had problems with releases but did not foresee a return-on-investment from implementing them.

Not all sub-practices were used. We consider that to be due to special characteristics of our student projects. They go on for only six weeks, so they do not encounter big problems if they do not {Write proper commit comments} or {Trace changes to stories}. Furthermore, the application is small and the stories are well defined, so they can ignore the consequences of not doing properly {Impact analyse refactorings} and {Impact analyse stories as part of Planning Game}.

6. Conclusions

In this paper, we have shown that some of the concepts and principles, that are very fundamental to SCM and seemed to be absent from XP, are indeed present. They are called by different names (like stories instead of change requests and Planning Game instead of change control board), but the contents and philosophy remain the same.

From our analysis it appeared that those XP practices that could seem dangerous and impossible from an SCM perspective: Collective Code Ownership and Frequent Releases, are indeed practicable. To a large degree they are supported by the standard XP practice Continuous Integration, but there are some additional requirements that have to be implemented for those two practices to be safe from an SCM perspective.

Furthermore, we showed in the analysis that some of the SCM requirements, both from a traditional and from a developer perspective, are indeed implemented by the existing XP practices. For those requirements that are not implemented by the standard XP practices, we have designed a number of SCM sub-practices that will ensure their implementation and we show how the added sub-practices together with the standard practices support and require each other.

Thus we can conclude that, when our new SCM sub-practices are added to XP's standard practices, XP is a both sound and complete development method with respect to SCM requirements. Our experience with using a sub-set of our new

SCM sub-practices on XP projects confirms that. Some of the new sub-practices were not used initially, but adopted after they suffered the consequences. Other sub-practices were not used at all, due to these XP projects being small and having a very short lifetime.

For the upcoming iteration of XP projects we plan to give all students knowledge and experience with sub-practices {Physical audit in release process} and {Automate and optimise the release process} through compulsory spikes. In addition, we have plans for investigating further sub-practice {Incremental refactoring} and its interactions with versioning in collaboration with a couple of individual groups.

7. References

[AB01]: Ulf Asklund, Lars Bendix: *Configuration Management for Open Source Software*, Technical Report, R-01-5005, Department of Computer Science, Aalborg University, Denmark, January 2001.

[Babich86]: Wayne A. Babich: *Software Configuration Management – Coordination for Team Productivity*, Addison-Wesley, 1986.

[Beck99]: Kent Beck: *Embracing Change with Extreme Programmin*g, IEEE Computer, October, 1999.

[Beck00]: Kent Beck: *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.

[Berlack92]: H. Ronald Berlack: *Software Configuration Management*, John Wiley & Sons, 1992.

[Berliner90]: Brian Berliner: *CVSII: Parallelizing Software Development*, in proceedings of USENIX Winter 1990 Conference, Washington D.C., 1990.

[chromatic03]: chromatic: *Extreme Programming Pocket Guide*, O'Reilly, 2003.

[HBM03]: Görel Hedin, Lars Bendix, Boris Magnusson: *Introducing Software Engineering by means of Extreme Programming*, in proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, May 3-10, 2003.

[JAH01]: Ron Jeffries, Ann Anderson, Chet Hendrickson: *Extreme Programming Installed*, Addison-Wesley, 2001.

[Leon00]: Alexis Leon: *A Guide to Software Configuration Management*, Artech House, 2000.

[MP97]: Tim Mikkelsen, Suzanne Pherigo: *Practical Software Configuration Management – The Latenight Developer's Handbook*, Prentice-Hall, 1997.