Configuration Management for eXtreme Programming

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

Abstract

Extreme programming (XP) is a software development method that prescribes the use of 12 different practices. Four of these practices (collective code ownership, continuous integration, small releases and refactoring) can indeed be given good support by the use of simple configuration management (CM) techniques. We report on our experience in providing many groups of novice developers with CM education, processes and tools to support the four CM-related XP practices in their projects. True to the spirit of XP both education and processes are very lightweight and we found that it was sufficient to focus on those CM aspects that are related to co-ordination and release. Furthermore, we found that even a simple CM tool will do as long as it supports the copy-merge model to allow and support parallel work. Overall none of the four CM-related XP practices caused any particular problems to the developers. In our case, the developers were students, but we claim (and argue) that the majority of lessons learned can easily be transferred to an industrial setting.

1. Introduction

There is an increasing interest from both industry and academia for Extreme Programming (XP) [Beck99] as a software development method for small groups. It is one of the so-called agile methodologies and prescribes the use of 12 practices. However, to make XP successful, its practices have to be carried out. Many people that are curious about using XP are uncertain if its practices can actually be carried out by other than its inventors – and how to make the practices work. Four of the practices that XP prescribes, Collective code ownership, Continuous integration, Small releases and Refactoring, have relations to Configuration Management (CM). In this paper, we focus on our experience with how simple CM techniques can help to support carrying out these practices and thereby contribute to the success of an XP project.

At the computer science department at Lund Institute of Technology, we have been running many XP projects the past two years. In this paper, we investigate those projects and extract our experience to say something about what level of CM knowledge, processes and tools are needed to support the CM-related aspects of an XP project. The goal of this paper is not to say whether XP or its practices are good or bad. However, it is to investigate if it is possible to actually carry out the CM-related XP practices and what has to be done to enable the developers to carry out the practices.

We explain how we have educated the students in CM concepts and techniques before their projects, and by reflecting on the lessons learned, we can indicate what kind of CM knowledge has been sufficient for our XP projects. Furthermore, we expose the CM processes that the students did actually use during their projects, and by reflecting on the lessons learned, we can indicate what kind of CM processes are suited for and needed on XP projects.

It is true that our student projects are scaled down XP projects. They run for a limited period of time and do not produce very large quantities of code. However, that does not have any significant effect on the validity of our findings. In fact, we can claim that some parts of CM even are stress-tested more in our projects than in "real" XP projects, as we have relatively many people working on relatively little code, thereby increasing the problems that could be caused by practising Collective code ownership and Continuous integration.

In general, we find that the four CM-related XP practices can actually work. There are no particular problems in carrying them out given the proper preparation and support. We do not claim from our experience that this is the only way – or even the optimal way – of supporting the CM-related XP practices. We do, however, say that this is one way and that we have experienced that it works.

In the following, we will first briefly describe the context of our XP projects, our methods for collecting data and XP and its four practices relevant to CM. In section 3, we explain how we teach CM concepts to the developers to give you an impression of their level of knowledge before they start their projects. The CM-related process that the developers use in their projects are described in section 4. In the following section, we detail the lessons we have learned and discuss to what degree they transfer to an industrial context. Finally, we draw our conclusions in section 6.

2. Background

During the last two years, the department of computer science at Lund Institute of Technology has used XP in a team-programming course for undergraduate students to introduce software engineering [HBM03]. The course has been divided in a theory part to teach the 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 XP best practices and the lab sessions introduce techniques and tools to support the practices. A written exam ends the theory part of the course to ensure that the students have sufficient knowledge of the methodology before the project starts.

The project part of the course is carried out in 12 teams with 8-10 developers. Each team also has two coaches who guide the team members through the development process. The team is assigned a customer that requests a product and supplies requirements through user stories and gives feedback on the currently released product.

The project is divided in 6 iterations. Each iteration consists of 2 hours of team planning, 4 hours individual spike-time, and 8 hours team program development. The 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].

To gather information for this paper the authors have analysed a total of 24 XP projects, divided equally over the two years the course has been given. Two of us have first hand experience from team coaching, while one has been an (partially) on-site customer. Besides participating in the actual projects we have held informal interviews with a majority of the coaches and analysed a survey regarding CM experiences taken by the team members. The written exam ending the theory part as well as a quiz prior to the lab session has also been examined to understand the students' maturity CM-wise prior to the project part of course.

XP is a lightweight methodology centred around 12 best practices [Beck00], e.g. pair programming, test-first, and simple design. Development is typically organised in short iterations with planning meetings in-between. An on-site customer is always available for informal discussions. Release rate is frequent to get rapid feedback from the customer. The rest of this section will focus on the practises that are CM related, i.e. Collective code ownership, Continuous integration, Small releases, and Refactoring.

One of the practices prescribed by XP is called *Collective code ownership*, which means that everyone owns all parts of the code. There is no notion of assigning ownership of code to individual developers or pairs and having them be responsible for that code. The whole team owns all of the code and is collectively responsible for it. 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. XP argues that the pair that finds something in need to be changed is also the best to fix it as they already have their head into it. If they need additional information to carry out the change, they can look up the pair that made the code in the first place and discuss the change with them. To ask someone else to actually make the change will only mean that you have to wait for that to happen (and in the meanwhile work around the problem) – and that they will probably not get it quite right the way that you wanted it anyway. When pairs work the code to implement their stories they will always find something that can be improved and it is the responsibility of the pair that discovers that to fix it, not the original creator of the code. When a pair has finished a task or a story they release their code to a common repository to make it available to all the others and thereby encouraging the others to take advantage of the collective code ownership to use (by continuously integrating it into their work) and improve the code. 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.

The practice of *Continuous integration* means that new code has 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 in into the pool of common production code. Such testing would ideally be carried out in a "clean" environment (possibly on a separate integration machine) and build from scratch to avoid "noise" from possible temporary files and to make sure that what you will integrate later is also what you really test. In case others have "released" changes to the common repository after you integrated it into your workspace, you will have to start all over again. However, "releases" will be rather infrequent even in an XP project and you will spend only short time on the testing, so it will happen only on rare occasions that you have to repeat your testing work. The idea behind continuous integration is to have everyone working always on the most current code, which will tend to make the code converge instead of diverging because divergent code is discovered immediately. Thus the more often pairs "release" their

code, the less likely it is that the team's code will diverge. Furthermore, it will avoid most conflicting merges to "release" often. Continuous integration also means that a pair should continuously integrate the latest version of the common production code into their workspace. Again the purpose is to discover problems early, to keep the code convergent, to ease later integration into the repository and to avoid troublesome merge conflicts. Being in sync with the rest of the pairs also makes it easier to fix "bugs" before the modules get too much out of sync, thus fixing fewer "bugs" at a time and having the code and changes fresh in mind.

In XP they also practice *Small releases*, which means that they try to put the system into production as early as possible and even before it has solved the whole problem. They also do releases very frequently, which can be anywhere from a daily to a monthly basis. What is 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 rationale is that this will provide early and frequent benefit to the customer in terms of delivering real business value. It will also provide early and frequent feedback to the developers so they can learn what the customer wants. The team and the customer can use the small 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. It will be a little more expensive up front, but in the long run it will be worth the money.

The practice that is related to CM, and probably the most tricky one to use, is *Refactoring*. 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. In XP the development is focused on coding and the design evolves after coding rather than up front prior to the implementation. 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. The goal is to keep the code as simple as possible and to improve the design while preserving the functionality. 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. This way it is possible to change the design on the fly when needed, so you are not locked in with early design decisions.

3. Teaching CM for XP

In this section, we describe what we did to bring the novice developers to what we consider a sufficient level of knowledge about CM techniques and processes to have a solid foundation for starting their XP project. We do not in any way claim that this is all they need to know, but given the restriction that we had just one two-hour lecture and one two-hour lab exercise, we found that this was the basics. For the rest they had to further educate themselves as part of the spike time available in the projects.

The use of CM in an XP setting is a partial goal of a larger course that introduces software engineering by means of XP. That makes the time for teaching CM quite limited. We have chosen to let one, in a series of six 2-hour lectures, focus on CM to support selected XP practices. The lecture is placed early in the course as the first in-depth study of some of the XP practices, preceded by only a brief introduction to the 12 XP practices and software engineering in general.

The lecture is supported by one 2-hour lab exercise to increase the understanding of the practices and to introduce tool support for basic CM 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 CM-related, they do, however, increase the student's skill in using CVS.

The lecture gives a general motivation for using CM techniques and introduces some of the goals of CM. It mentions the different CM roles in an organisation and goes on with detailing the role of the developer. The main emphasis is on the problems of team co-ordination and describes the double maintenance problem, the shared data problem and the simultaneous update problem [Babich86]. The general CM principle of immutability is introduced and explained, as are the concepts of version group and version group structure. They are introduced to the CM problems of interaction between the team and the individual and taught the concepts of repository and workspace in that context.

To deal with the above-mentioned problems of team co-ordination a model, copy-modify-merge using strict long transactions, is presented and its support for the parallel work in a team, that is a consequence of practising Collective code ownership and Continuous integration, is discussed in length. In that context, they are also introduced to the concepts of merging and diffing and the use of a log to create a history of changes.

The literature used for the CM lecture is chapter 1 of [Babich86] and chapters 11, 15, 21, 22, 7 and 8 of [JAH01].

Prior to the lab exercise the students have done a tutorial to get acquainted with CVS and also studied further material concerning the copy-modify-merge model [Andersson03]. The CVS tutorial explains how to connect to a central repository, how to use diff and basic understanding of merge. Traceability and logging is also explained and their use motivated. The lab exercise is preceded by a short written test to verify that the students understand the central concepts and have basic knowledge

of CVS syntax. That way the lab can be used to gain a deeper understanding of the underlying model instead of focusing on tool syntax.

A lab exercise, supervised by an assistant, is a group of 20 students that are divided in 10 pairs, where each pair plays the role of two developers working in parallel. Each pair solves a set of exercises that result in merge situations of various complexities. An exercise is stated in general terms and the pairs are expected to use their very basic knowledge of CVS to solve that practical problem. Each exercise is supported by a number of questions requiring the students to discuss the behaviour and reflect on the expected behaviour.

Two subsequent lab exercises deal with Test first and Refactoring. In the Test-first lab, students also have to work in pairs and we require them to use the repository and the update-commit process taught in the CVS lab exercise during their work on the assignments. This is also true for the Refactoring lab exercise, where they use the tool Eclipse [Eclipse] for doing small refactorings, still following the update-commit process using Eclipse's graphical user interface to CVS.

We have been through two iterations of the XP project-course and the literature and lecture on CM has remained practically invariant. For the lab exercises we made some adjustments based on experience from the first year. What is described above is the adjusted lab exercise set-up that was used the second year. The first year there was no lab exercise on Refactoring and the CVS lab exercise came before the Test-first lab exercise. As a consequence the students had too little practical background with CVS when they started using it in their projects. Furthermore, we found that the material we had given the first year as preparations for the lab exercises (section 1.1 of [Fogel00]) did not work well and decided to have something written that was entirely focused on our particular setting [Andersson03].

In section 5, we elaborate more on the lessons learned as a result of the experience of teaching CM concepts and techniques to novice XP developers.

4. Practising CM in XP projects

This section describes how the students used the CM knowledge acquired during the lecture and lab sessions in the project part of the course. The students had been strongly advised to follow the frequent update-merge process and had all practical experience using the tools from the lab sessions. Apart from that they were on their own – and some did not even take our advice (often resulting in huge merge conflicts). There was no formal CM education at all during the project apart from what the teams decided on their own. Most teams used some spike time to gain further knowledge of CM (primarily the CVS tool). This sometimes resulted in tutorials or checklists used by all team members.

The teams used four, more or less, formalized processes related to CM during the project: normal work (implementing a story while updating and synchronising their work with a repository), spikes (carried out in-between development sessions), refactorings (resulting in somewhat different change behaviour), and the release process (practiced every other iteration).

Normal work process

The normal work process for a pair starts when they are assigned a story/task to implement. The pair then creates a workspace and makes sure it is up-to-date with the central repository. They start implementing the story using the test-first methodology. During their implementation they frequently update their local workspace to stay in sync with the repository. When the story is fully implemented, all unit- and acceptance-tests pass that is, the changes are integrated. Strict long transactions are used to enforce that all code in the repository is tested in the local workspace prior to integration. Commit comments are used to explain why the changes were integrated. By doing frequent updates the possible merge conflict are usually simple to solve manually. This experience was often learned the hard way after an initial long implementation session without update, resulting in much more complicated merge problems.

Work during spike time

The spike time is used to experiment and to gain knowledge needed to implement some future story. Production code must not be implemented during spike time, as spikes are usually not done in pairs. Unreviewed code produced by a single person would be dangerous to include in the product. Each team used a separate repository dedicated to spikes. The result from a spike was usually a document or a small set of source files, not dependant on the developed product. Therefore, a separate repository was usually not a problem. When a spike was depending on some production code, a separate branch in the production repository, later to be discarded once the spike is done, could probably have been used successfully instead of copying code to the spike repository and rely on non synchronised code.

Refactoring

The students learned the hard way that large refactorings during program development creates huge merge conflicts. The nature of a refactoring often results in changes scattered all over the code whereas a story often is quite local and merely adds code. As they were not able to follow the recommendation of dividing a refactoring in small (reversible) steps, most teams adopted

the fallback strategy to do it during spike time when no one else worked on the normal repository. That only partially solved the problem as the other team members had to start the programming session by solving tedious manual merge conflicts. Several teams used the planning sessions to schedule primarily stories that would not affect the same code base as larger refactorings (particular changes to data structures) and to increase awareness and promote even more frequent updates.

Release

All they knew about how to do a release was a tiny bit from the lecture. So all teams had to spike a release process. All teams made a disastrous first release; some releases did not even work. And it took far more time than anticipated. However, after that first try the two extremes were that some remained there, whereas others gradually progressed to an almost automated release process that took only a few minutes to carry out and ran without problems. This was the CM activity that differed the most among the teams. A general conclusion was that teams using an automated release process were far more successful that the teams using manual check lists or no release process at all.

The normal work process was carried out successfully and all teams concluded that they could not have completed the project without some kind of support for parallel development and synchronization. The few problems that occurred were mainly slow merges due to not frequent enough updates and synchronisation, mostly during refactorings.

The teams started working on a very small initial code base. Thus, in the beginning of the project they were using the merge facility of CVS to the extreme. There will probably be less of that in an industrial setting starting on a larger code base or gradually increasing the team size to fit the current project code base. The use of very short iterations and direct feedback from the customer on the current product probably simplify the use of the repository. Using long iterations and delayed customer feedback would require release branches and retrieval of older versions.

The release process was naturally the CM activity that differed the most among the teams, as they had to create it themselves. Automated tasks and a formalised release process were crucial to the teams that were successful in the frequent product release to the customer.

The overall impression is that the process is well understood by the students and that they will choose to use it in future, smaller project-courses.

5. Lessons learned

Here we extract the lessons we have learned along the way that is going from year one to year two. Furthermore, we reflect on the lessons that can be learned after year two. In addition we reflect on to which degree our experience is bound to a student setting or whether and to what degree our set-up and lessons can be transferred to an industrial setting and, if not, how it might differ.

Lessons learned can be categorised into three types: educating XP developers in CM matters, providing CM processes and methods for XP projects, and the requirements to a CM tool for an XP project.

Educating CM

Most of the knowledge the students need belongs to informal CM [AB02] and is related to CSCW (Computer-Supported Cooperative Work), as it is co-ordination of a team effort. More concretely this means that the education should deeply root the copy-merge (update-commit) model in their minds so they just do it without even thinking about it (just like breathing or walking). Our experience is that one lecture and one lab are sufficient to give them that knowledge and thus to prepare them for the project – though we profit from practical use of CM also in the two succeeding labs where the focus is on test first (unit test) and refactoring. To retrieve knowledge needed of more formal CM, as for example the release process, most teams used spike time. Thus, formal CM was learned more on demand and, often, as a consequence of a previous failure. E.g. almost no teams made a proper first release, but most teams learned from this to the following releases. However, even though the final release was well functioning almost no teams used techniques to optimise the release process, e.g. using branches, scripts, etc. Should we have also have taught them how to use branches? As always, there are pros and cons; We have found it important that they do get very familiar with the basic model before we introduce something new. Secondly, we have to respect that they have a 40-hour week (i. e. there is a limit to what we can put into one lecture and one lab – and that is what we have and will ever get). On the other hand, one extra hour of lecture should give them sufficient basis for better spiking the release process, which should give a higher starting level for the first release process and a steeper learning curve. As a comparison, we use less than four hours on all four CM activities in a dedicated CM course taught to senior undergraduate students [AB03], so one extra hour should be enough.

In our course, with only 6x2 hours of lecture we may experiment by giving some groups this extra hour of teaching branches. In an industrial setting we recommend to always include this hour.

We have to point out that even though we use time-budget, our students are pressed for time and very focused on the cost-benefit factor of what they are doing/adopting.

The reason for not learning a more cost-effective release process is that this process only will be used twice. In a real setting, planning for many more releases, of course an optimised release process has a better cost-benefit factor.

Not using branches also affects how to manage spikes. To store spikes made and to share knowledge about spikes a spike repository was used. In this repository any new technique or change can be tried out without affecting the production code. However, this repository soon gets out of sync with the production repository. Many experiments will not be relevant if the "context" of the experiment in not up-to-date. Maybe a better way would be to use branches in the production repository. Branches can easily be updated merging from the production (main) branch. However, using branches also makes it tempting to merge from the spike branch to the production branch, which is *not* allowed.

The changes made to the lab sessions for the second year resulted in a more solid understanding of the update-commit model. The emphasis on a general model even during the lab session instead of a specific tool actually decreased the complaints on the tool and increased the appreciation of the given support.

XP projects

During programming time all awareness and co-ordination is created by being in the same room, using oral communication and/or using a whiteboard to keep track of who is working on what. This means that no other tools or processes were used to obtain awareness. One consequence of this is that commit comments are rarely read although properly written. Instead only the time stamp and information about who committed the change is read, and that person directly contacted. The fact that they are located in the same room decreases the need for long commit comments about the changes made. However, they probably would have read the commit logs if they needed to remember why they did something in the past. Now the duration of the project was sufficiently short that they could remember when someone else asked them.

Students very rarely retrieved older versions and seldom used tags. The first is quite natural as XP "doesn't look back" – the second is probably more due to the "student" nature of the projects (no maintenance etc.). However, both things are needed during larger projects in the occasions where an older version or release is really needed.

Private versioning was not used. It could have led to abuse of the repository – but if really needed should have been done by branches. Branches could also have been used for spikes instead of the spike-repository (but, if used, one could be tempted to merge the branch (spike) back into the main line).

The fact that they have to carry out unit-testing very often, means that they need fast builds to allow a rapid code-build-test cycle that actually works.

They were very weak on using branches, but probably they found that it was not worth the effort (or they would have liked us to support them with a lab – being afraid that they were not able to stand alone).

The students noticed that collective code ownership (+ continuous integration) means that they needed coding standards in order not to get silly merges or merge conflicts (just like the problem when Emacs and Eclipse format the code in different ways).

Refactoring is the killer in this game – and to some degree also stories that have the same properties as a refactoring (touching large parts of the code and taking a long time to do). Often one refactoring affects almost all code, thus resulting in merge conflicts with all the other pairs. Refactorings will probably always be different from stories in that they often take away (or change or move) code, whereas stories almost always just adds some new code – and this is what makes them so hard CM-wise. Even though refactorings may be a problem also in "real", larger XP projects, our reality with a rather small code base and a compact, 8 hour, coding phases increase that merge problem. The lesson learned is that refactorings either should be done outside the long-labs (so as not to block other pairs but may still cause problematic merge conflicts) or more effort in doing refactorings in small steps is necessary. Apparently the latter was quite difficult for the students, so they don't do it that way. We also emphasise the importance of a frequent update/integration cycle during refactorings.

CM tool support for XP

XP has very few requirements to the CM tool. Most importantly it should allow the pairs to work in parallel – which is hard to do (especially in the start) if it does not support a copy-merge and update-commit models. It should also include a very strong merge facility. We used CVS, which has a very clear update-commit model easy to learn and understand. However, its support for merge is weak only supporting two-way merge. Almost all larger CM tools have better merge support, but are often more complex and then not that easy to understand in only one two hour lab. Our use of Eclipse last year, that improves the usability of CVS through a GUI and some initial support for refactorings, was appreciated by the students but was not considered essential. A GUI is probably more needed if branches were used more frequently though.

In addition to this required basic functionality, there should be support for awareness like tracking of authors of changes - some people also used the watch functionality. Interestingly plain version control seems to be less important and will be there in even the simplest tool. As mentioned above, due to Team in one room

awareness and co-ordination was done mostly through oral communication and not through the CM tool, e.g. using the CVS watch functionality (which, besides, is improved in the Eclipse environment). Only a few teams experimented with the watch functionality.

One limitation in the CVS tool (and that applies to most tools) is that it handles merge for ASCII text lines only. That means that you can forget about writing the user manual in Word, that uses a proprietary format – or use Netscape Composer, that constantly reformats the contents, as the editor for the documentation. But that is in general – as these things do not merge easily – and not only in XP projects (though some XP practices causes us to do more merges).

Another weak point of almost all tools including CVS is their support for refactorings. Renaming a file and moving a file changing the file hierarchy is nearly not supported at all. However, also in this respect Eclipse has relative strong support. In larger XP projects involving continuous refactorings such support is invaluable.

It should be noticed that the teams used CVS merely as a merge tool and hardly at all as a versioning system. As discussed above one reason can be the "student" nature of the project with no maintenance but we emphasise that the support for parallel development and workspace synchronisation are the most used features of the tool.

6. Conclusions

Our overall conclusion is that the four CM-related XP practices (Collective code ownership, Continuous integration, Small releases and Refactoring) are indeed practicable in real life by novice developers at an early undergraduate level. It is possible to provide support for the practices by using simple CM techniques and methods.

The necessary CM education is not very expensive. For novice developers, like our students, one 2-hour lecture and one 2-hour lab exercise in using the CM tool is a sufficient foundation to make them survive at the start of the project – and learn more by themselves as they go on. Within that time-span we can focus on team co-ordination for the theory part and the update-commit routine for the practical part.

The CM-related process that was used most during the project was the update-commit to support the copy-modify-merge model. They also practised frequent updates to allow for Continuous integration into their workspaces. Some learned this the hard way from getting difficult merge conflicts because they waited too long between updates in the beginning. Once they had adapted to frequent updates they got no, or only simple and easy to solve, merge conflicts. The release process was at first very chaotic for all groups due to the lack of teaching focus on that aspect. However, quite a few of the groups progressed to almost automated releases from studying on their own and doing four releases during the six iterations. There were no formal or supported processes for creating awareness in the group. Being in one room all the time meant that they would rely on informal communication to create sufficient awareness. In an industrial setting, where it may not be possible to practise Team in one room and where projects run for a longer time, this may not be practicable. Finally, there is the process for doing Refactoring. This created problems for all groups as it resulted in big and complex merge conflicts. This was due to the fact that they all did the refactoring in one big step. Some groups refrained from doing more refactoring after that experience; others did refactorings outside of the programming time as a spike. However, the best way to avoid the problematic merges is to do refactorings by the book as small steps. This is the process that should be adapted by industry and not that of our students.

As for the requirements to a CM tool to support XP, they are very few. Firstly, our students almost never used the CM tool as a versioning tool. Very few occasionally used branches, tags, and diff or retrieved an old version. That might differ somewhat in an industrial setting with longer-lasting projects. However, we still find that the traditional versioning requirements will be satisfied by just about any CM tool. Secondly, our students saw the CM tool as primarily (or only, in some cases) a merge tool. This means that the merging capabilities – on both file and structural level – is very important. Finally, refactorings sometimes created problems for the CM tool. In CVS, it is impossible to rename or move files and retain complete history and traceability. This is, however, not just a deficiency of CVS, but something that most CM tools suffer from.

In total we can say that just like the XP method and practices are said to be light-weight, so goes for the necessary CM education, processes and tool that can support XP – and indeed other methods based on tight team collaboration.

We are just about to start a third round of XP projects. For that we have plans to carry out experiments with selected groups to gain more experience. In particular we want to focus on the release process, the use of branching and version-aware refactoring.

7. References

[Andersson03] C. Andersson, "Introduction to Configuration Management with Concurrent Versions System", Department of Computer Science, Lund Institute of Technology, Sweden, Spring 2003.

[AB02] U. Asklund, and L. Bendix, "A Study of Configuration Management in Open Source Software", *IEE Proceedings - Software,* Vol. 149, No. 1, February, 2002.

[AB03] U. Asklund, and L. Bendix, "A Software Configuration Management Course", in proceedings of the 11th International Workshop on Software Configuration Management, Portland, Oregon, May 9-10, 2003.

[Babich86] W.A. Babich, "Software Configuration Management: Coordination for Team Productivity", Addison-Wesley Publishing Company, 1986.

[Beck99] K. Beck, "Embracing Change with Extreme Programming", *IEEE Computer*, October, 1999.

[Beck00] K. Beck, "Extreme Programming Explained: Embrace Change", Addison-Wesley Publishing Company, 2000.

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

[Eclipse] The Eclipse Java Development Tools subproject, http://www.eclipse.org.

[Fogel00] K. Fogel, "Open Source Development with CVS: Basic Concepts", http://cvsbook.red-bean.com/cvsbook.html, 2000.

[HBM03] G. Hedin, L. Bendix, and B. Magnusson, "Teaching Extreme Programming to Large Groups of Students", *Journal of Systems and Software*, in publication 2003.

[JAH01] R. Jeffries, A. Anderson, and C. Hendrickson, "Extreme Programming Installed", Addison-Wesley Publishing Company, 2001.