Djupstudie EDA270 - Agile Software Configuration Management

Andreas Back d01ab, Ola Bodin d01ob

22nd February 2006

Abstract

This article presents the results of applying some of the practices defined in the article "SCM practices for XP teams". These practices aim to prove that Extreme Programming can fulfill the requirements Software Configuration Management states for a development process in order to be complete.

Contents

1 Introduction

This in depth study is regarding the proposed additional practices to extreme programming presented in the [1]"SCM practices for XP teams" article. The article tries to bridge the gap between the XP software development process and the demands the traditional SCM practices identifies as necessary for a fully compliant development process by introducing additional practices.

We have tried to figure out to what extent these practices actually work, if they are necessary and whether they still keep the agile development process agile or if they burden the process.

In order to fully understand everything we think it could be necessary to read the [1]"SCM practices for XP teams" article prior to this one as we our definitions of SCM and its practices are somewhat brief compared to the original ones.

In section 2 we give a overview of what SCM is and how it relates to agile programing methods. In section 3 we describe the proposed extensions of XP and give our comments to these. In section 4 we discuss which of the practises we find necessary and which we don't se a need for. At the end in section 5 we summarize our findings.

2 Background

In order to fully understand the reason for this article we believe it to be important to understand the different views on software design or software architecture that exists. Traditionally the way of an engineer to produce something, whether it is buildings or mechanical parts is to carefully analyze the problem. Following is a design and architecture phase trying to cover every possible problem that might arise during production. The reason for this approach is that the cost of correcting a fault traditionally is believed to rise considerably worse then linear as longer into the project. And thus identifying as many possible problems early on is the economic foundation for a successful product.

This attitude has also made it's way into software related projects as it is established in many other areas and is believed to guarantee a quality release at a minimum (or at least affordable) cost. It's neatly defined as software engineering.

One thing that possibly differs software development from any other engineered practice is that the actual cost of production (ie producing and packaging a build) is much lower. This has inspired other attitudes to software development, for instance the agile methods.

In agile methods the cost of changing requirements or identifying that the product differs somewhat is believed to be much lower and affordable. By lowering the amount of carefully thinking stuff through (in the meaning of spending several months on simply planning the software product) and being prepared to always change the product in any way that it needs the belief is that a fully working quality product can be achieved in much less time, costing less and giving the developers a more endurable pace.

The practice of software configuration management, SCM, is about adding traceability to all parts of the process. It is about making every version of every important part that the project consist of (the configuration item) registered and available for traceability and identifiability. This enables the team to revert to an older, perhaps working version if something breaks or the need for a patch to that version arises. As well as the ability to from the configuration item itself identify which version of the product it relates to.

This applies to both traditional software engineering as well as a more agile approach yet the clear definitions and defined process makes if fit better with the first one.

This article tries to look at methods for adding explicit SCM practices to the agile programming method eXtreme Programing (XP). The practices are proposed in the article mentioned above[1]. The article tries to establish a set of additional practices to make the XP process complete from a SCM point of view. This in-depth study tries to define how well a team adapts these practices as well as if the actually are needed. We will now cover each proposed practice and provide our comments or findings about them.

3 The Practices

In this section we define the SCM practices proposed in the SCM practices for XP teams[1]. For each practice we state our thoughts and any experience about it with regards to our experience as coaches during the EDA270 course.

3.1 Incremental Refactoring

Making large factoring's in one go prohibits continuous integration and makes it harder to return to a working version without loosing to much work. In order to address this the article proposes the XP team to divide up the larger refactorings into several smaller ones that can be performed and committed individually.

So far we've had experience with this on one occasion where a very large structural change that impacted most of the program were about to take place. The refactoring was actually divided into two separate refactorings done by two pairs. This proved to be successful. As most of the other stuff, the rest of the team was about to implement, was affected by this any reduce in time it took to implement was greatly appreciated. The longer period between commits the larger the chance is of a large merge conflict. And by dividing the refactoring into several incremental once commits are done more frequently.

This practice is of curse important in all software development whenever you want to do a major refactoring without stooping all other development. The only other way to have some people continuing the development while others do the refactoring is to branch but if it is a major refactoring of central parts of the code the merge can be horrible.

3.2 Impact Analyze Refactorings

Large refactorings should be impact analyzed with regard to possible merge conflicts prior to implementation to reduce the number of actual merge conflicts and to increase the teams awareness of ongoing refactorings.

Though we generally tried to keep larger refactorings out of the actual lab hours during the project by assigning them as spike's once identified some large refactorings had to be implemented at once. Our experience is that taking the time to discuss the proposed refactoring thoroughly within the pairs affected in the group pays of greatly as the team members then are aware about which parts of the code that are about to be changed.

This practice may seem to be very obvious but the fact that these analyzes are very hard to do justify that this has its on practice. Before you do a refactoring you almost always try to think about what this will mean in the code, what will have to change if I do this but still you often bump in to crucial consequences that you still missed. To really think through the refactoring before making it will help to minimize this misses.

3.3 Use a copy-merge work model

The article proposes the copy-merge repository work model to be used within XP teams as all members then have their private workspace for implementation. All tests can then be performed before committing to the collective repository.

This does seem reasonable, if for nothings else simply that a private workspace is needed in order to keep the changes out of the repository until they have passed all testing. We have no actual experience in working with any other repository tool then cvs within teams. If you want to embrace collective code ownership then you will have to have a way of merging different peoples code and you will have to have your own copy of the code to write in and run your test on. I have a hard time to see any other work model to use if you are using XP so this practice is definitely justified.

3.4 Impact Analyze stories as part of the planning game

This practice points out the importance of impact analyzing the stories during the planning game and including the cost of possible merge conflicts in the total cost estimation for the story.

We believe that most (if not all) experienced XP teams do this naturally, more or less continuously when time estimation stories during the planning game. However another important part of impact analyzing the stories is that many stories are somewhat dependent on each other, or that stories will cost a lot less if other stories are completed before.

There is though a possibility that adding this practice to the planning game turns the agile process into more of a hybrid one even if the amount of analyzing is kept to a minimum. We have limited experience within our XP team of actually formally doing this and we would like to discuss this with the rest of the coaches and teams but it is our first impression that the team found this to be non needed formality.

3.5 Physical Audit in the release process

A Physical audit should be done for each automatically generated release to ensure that small errors does not slip through. For instance missing configuration files or modules.

Our experience of this is that it is important and a very cost efficient way to ensure that the release actually is working. Simply unpacking the release, installing it (favorably on a fresh set of hardware) and trying to start it will catch many simple mistakes not though about during the specification of the automated release scripts.

This practice is of curse not special for XP, every time you make a release in any type of project it is a good idea to make a physical audit. Someone in the developing team should try the same things the customer will do. Since XP dose not have a formal testing phase this physical audit is easy to forget, in a more traditional waterfall method this is a natural part of the testing phase.

3.6 Define configuration items and their structure

What items that are configuration items and the structure for them should be decided early in the project.

If one is to use cvs as the repository this is crucial due to the lack of features regarding properly moving and deleting files. Though this is a problem with the tools seen from a XP perspective and the team should not have to find ways to work around this problem, rather a more supporting tool should be used.

If the code should be agile and always be the simples design appropriate for the current situation we can't see why not the structure of the repository could be more agile to. Of curse a initial structuring of directories for code, spikes and other documentation is needed but as the project evolve the repository should be able to evolve also.

3.7 Trace Changes to stories

This enables traceability between implemented stories or tasks and the actual commits in the repository.

Traceability is something SCM values as very important, it's one of the main things of SCM activities. With traceability one can determine which changes, and in which files, that have been made to implement a certain story or task.

In traditional software development a change control board decides whether or not changes should be implemented and then by whom. So a record of what has been changed will be available. In XP however at best the log files for the repository as well as any documentation from the tracker would trace each story to the changes.

One possible solution would be to include the current story or task implemented in each commit comment. We believe that this is a small enough effort that it doesn't burden the agile process too much. This will be more effective if the team follow the practice of small changes that is tested an committed, by committing often the traceability to what story or task that was implemented will be increased.

3.8 Write proper commit comments

Each commit comment should properly document which changes that have been made and the reasons for them increasing the actual awareness within the team as to what has happened earlier.

We have currently told our group to include the current task they are working on in each commit comment, as to which extent they have complied with this we are not fully aware of but until today we have had no reason that we know of to actually use this information. We believe the reason for this the short time span of the project as well as the smaller team size in XP.

This practice relates to "Trace Changes to stories" in that with better commit comments it will be easier to do tracing. But this practice is hard to follow when doing small changes and committing often, if you don't make a big change the comment is hard to express in a meaningful way, and we think that this may be one of the reasons that commit comments ofter are quite badly written.

3.9 Automate and optimize the release process

Make it possible to automatically perform a release using scripting or other functions within the tools. This automation should package everything properly that is to be expected from the release and the end product should be the finished release itself.

Usually the release process contains many different things, many of which are easy to forget or in other way miss. For instance making sure configuration files or other files the release is dependent on really is packaged. By automating the release process a successful release is very probable given that one has been made prior. We cannot find any real argument for not automating the release as many tools exist that gives the functionality at a very low cost.

This is also a practice that allays not only to XP but to software development in general. If you are doing a larger project of more than fore people it is almost a necessity if the release is to be done in a reasonable time.

3.10 Use a version control tool

Use a version control tool to synchronize the common configuration items into a common repository.

The XP practices themselves rely heavy on tool support. Continuous integration would be very hard to implement without any tool support. The version control itself might not be necessary for all teams since XP much more then any other development process lies very little focus on the past versions. The merge support and conflict resolve support though that most of these tools provide are probably much needed. Any team that might face product updates or patches to any version other then the most current one would very much need some kind of version control.

3.11 Use a build tool

Use a build tool for building the executing product as well as producing the documentation.

Although we agree on the importance of using a build tool it's very hard to imagine anyone actually implementing a full XP development process without it. The cost is very low in implementation and the payoff is high for anything built more then once (manually building libraries used and keeping track of dependencies can be really tedious and takes time even for medium size projects). The more time one puts into configuring there build tool for there project the more they will be able to reuse and the more time they will save in the end.

3.12 Keep the repository clean

The common code in the repository should always work. Everything must compile and all junit tests should pass.

For any working XP team this should come naturally as the cost of non compiling or in any other way erroneous source code as well as the spread of it within a very short team quickly reaches impressive effects.

We did a test in our group where we let our team check in broken code due to a major refactoring they didn't finish at et end of the iteration. What we wanted to see was how hard this was to fix in the following week. The test went quite well the pairs that had coded the refactoring had time during the week to look at what needed to be done and was able to fix the problem early the following lab. The reason this went so well is probably because no one else was committing into the repository during the week so no one was affected by the broken code. Trading speed for security in this way is not the XP way of doing things but it's a gamble with potential high payoff, thou we would not recommend it if you don't have confidence that it wont slow you down.

4 Discussion

4.1 Practices we found needed

These are the practices that we believe are truly needed for a working XP process and of such importance that they need to be formally specified as practices.

4.1.1 Incremental Refactoring

By using incremental refactoring, integration can be done more continuously and in much smaller steps. This will be of no doubt to great help for any team and reduce merge conflicts.

4.1.2 Automate and optimize the release process

The difference in time and effort between a manual release process and a tool supported automated process is very large. Often the release includes small tweaks, or other stuff that can be easily ,forgotten simply because of the volume of them. This is especially true if there is limited time for the release. The effort that need to be put in producing any form of simple script supported automated release is so low that it will pay back without any doubt for any team.

4.1.3 Physical Audit in the release process

By physically auditing the finished release one makes sure that the amount of small errors causing the release not to run or function optimally is reduced. Small errors can for instance be missing files or similar. If the release process is optimized such problems should only arise the first time for each new dependency.

4.1.4 Use a version control tool

To successfully apply continuous integration one is dependent on tool support for merging different source version and handle, or at least indicate eventual merge conflicts. This is a feature which is normally found in common version control tools.

4.1.5 Use a build tool

There really are no valid reasons for not using a build tool. Any project with any form of dependencies will gain from it, even if it's just a simple script performing all the commands needed for the build.

4.1.6 Keep the repository clean

A clean repository without compilation errors is crucial for a working team. A full set of working acceptance tests likewise ensures that all team members tries not to break something, compare this to that its much easier to leave something broken if it was like that when you arrived. A update/commit intensive process as the XP process fully requires the repository to be clean in order to function optimally.

4.2 Practices we found not to be needed

Here are the practices we think are not necessary for the XP process.

4.2.1 Define Configuration Items and Their Structure

It feels kind of awkward if a process values agility within the source code but not in the file or directory structure itself.

We believe that the configuration management tool in use itself defines if this is needed or not. If one uses the CVS tool this practice is crucial, however if subversion is used one should be able to change the directory structure as they see fit throughout the project duration.

4.3 Practices we think are nice to have

These are the practices we find nice to have but that we have too little experience to actually state whether or not they are needed. They are all sound and some of them we think most successful teams are already performing, just that they are not specified as specific practices.

4.3.1 Trace changes to stories

The need for tracing changes have never really occurred at any point of the project and thus we have no experience about it. If the need arise though it could very well be very crucial and if no traceability is available great problems could very well be the result.

4.3.2 Impact Analyze Refactorings

Hopefully, anyone about to implement any refactoring takes a minute to reflect on eventual side-effects or dependencies. Though, applying this practice could be a very nice way of assuring that the whole team is aware, and have the possibility to add any insight, to refactorings about to be made. The risk of a large merge conflict should be reduced.

4.3.3 Impact Analyse Stories as part of the planning game

This practice depends alot on how much the team finds it. If it finds the practice to be working it could very well add valuable information to the team. If not it results in more work during the planning game. While the intentions where very good very few stories actually got any form of usable information in the project. From what we have heard this was the same for most other projects.

4.3.4 Write proper commit comments

The XP process is very focused on progress, not on previous work, thus what we have seen so far is that the log of the repository is used very seldom. When it's actually used itt's mainly to find out what team members that last committed something, in order to perhaps settle a merge conflict or ask questions about what they did. This is all covered without any commit comments at all.

The cvs repository tool have no support for grouping multiple file changes into one commit, and thus some people specify the collection of files modified in the commit comment. However this is a tool specific practice and with other tools this is not needed.

4.3.5 Practices we are unsure of

We are somewhat unsure about one of the proposed practices,

4.3.6 Use a copy merge work model

Any model that support the different team members to work in a separate sandbox should work, or does all of the possible solutions here adhere to the copy merge model? It feels awkward to state a specific model to be needed though it is certain that, in order for TDD development to work, a separate sandbox for each developer is needed.

5 Conclusions

Much work trying to get the agile and more traditional software engineering oriented developers to respect each other have resulted in solutions that the traditional developers can agree with but that the agile developers are not quite that fond of. Part of the presented practices are indeed helpful while others simply states what many (successful) teams are already doing. One should be very careful with adding formality to agile methods simply because thats heavily lessens the agile part.

However if a software development process is to be judged only by the documents describing it every part of it needs to be documented in order for it to be accepted as a fully SCM compliant process. Perhaps the most important answer the practices can give is that it actually is possible for an XP like process to be fully SCM compliant.

Being fully SCM compliant adds safety to the process, and any process relying on practices that are not stated (as some of the practices have been identified as being necessary) relies heavily on the team itself and success will then vary very much between teams.

6 Acknowledgments

We would like to express our deepest thanks to the programming team we had the fortune to coach during the spring of 2006.

References

[1] Ulf Asklund, Lars Bendix, Torbjörn Ekman: Software Configuration Management Practices for Extreme Programming Teams

[2] Lars Bendix & Ulf Asklund: A Study of Configuration Management for Open Source Software

[3] Shawn A. Bohner & Robert S. Arnold: Questioning Extreme Programming

[4] Pete McBreen: An Introduction to Software Change Impact Analysis

[5] Tom Milligan, IBM: Better Software Configuration Management Means Better Business : The Seven Keys to Improving Business Value

[6] Christian Rose et.Al.: Software Configuration Management Plan for the HolidayTree Project

[7] Lars Bendix, Otto Vinter: Configuration Management from a Developer's Perspective

[8] Peter H. Feiler: Configuration Management Models in Comercial Environments

[9] Roy Andersson, Lars Bendix: Towards a Set of eXtreme Teaching Practices