SOFTWARE CONFIGURATION MANAGEMENT IN AGILE DEVELOPMENT

Lars Bendix, Torbjörn Ekman
Department of Computer Science,
Lund Institute of Technology,
Box 118,
S-221 00 Lund,
Sweden
Email: {Lars.Bendix | Torbjorn.Ekman}@cs.lth.se

Abstract: *Software Configuration Management (SCM) is an essential part of any software project and its importance is even greater on Agile projects because of the frequency of changes. In this chapter, we argue that SCM needs to be done differently and also cover more aspects on Agile projects. We also explain how SCM processes and tools contribute both directly and indirectly to quality assurance. We give a brief introduction to central SCM principles and define a number of typical Agile activities related to SCM. Subsequently, we show that there are general SCM guidelines for how to support and strengthen these typical Agile activities. Furthermore, we establish a set of requirements that an Agile method must satisfy to benefit the most from SCM. Following our general guidelines, an Agile project can adapt the SCM processes and tools to its specific Agile method and its particular context.*

KEYWORDS: Software management, Groupware, Software Configuration Management, Agile, Team Co-ordination, Continuous Integration.

(1) INTRODUCTION

In traditional software development organisations, Software Configuration Management (SCM) is often pushed onto the projects by the Quality Assurance (QA) organisation. This is done because SCM in part can implement some QA measures and in part can support the developers in their work and therefore helps them to produce better quality. The same holds true for Agile methods – SCM can directly and in-directly contribute to better QA on Agile projects.

Software Configuration Management (SCM) is a set of processes for managing changes and modifications to software systems during their entire life cycle. Agile methods embrace change and focus on how to respond rapidly to changes in the requirements and the environment (Beck, 1999a). So it seems obvious that SCM should be an even more important part of Agile methods than it is of traditional development methods. However, SCM is often associated with heavily process-oriented software development and the way it is commonly carried out might not transfer directly to an Agile setting. We believe there is a need for SCM in Agile development but that is should be carried out in a different way. There is a need for the general values and principles of SCM, which we consider universal for all development methods, and there is a need for the general techniques and processes, which we are certain will be of even greater help to Agile developers than they are to traditional developers.

to appear in I. Stamelos, P. Sfetsos (eds.): "Agile Software Development Quality Assurance", Idea Group Publishing, Hershey, Pennsylvania, February 2007.

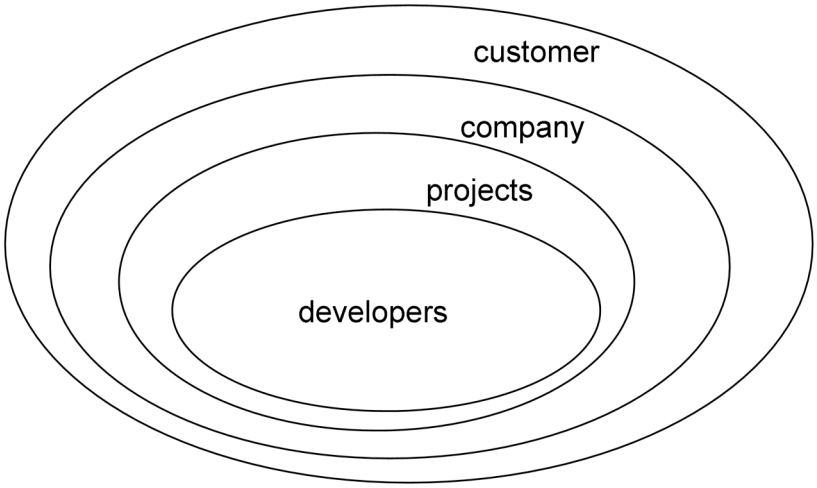

Figure 1. The different layers of SCM.

There are some major differences in Agile projects compared to traditional projects that heavily influence the way SCM can – and should – be carried out. Agile methods shift the focus from the relation between a project's management and the customer to the relation between developers and the customer. While traditional SCM focuses on the projects and company layers in Figure 1, there is a need to support developers as well when using SCM in Agile projects. Shorter iterations, more frequent releases, and closer collaboration within a development team contribute to a much greater stress on SCM processes and tools.

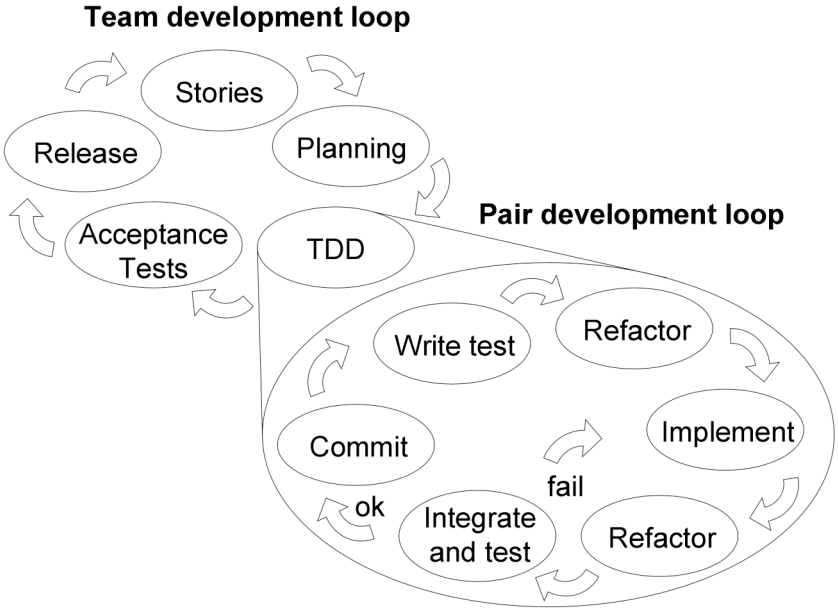

Figure 2. The main development loops in Agile.

Agile methods are people-oriented rather than process-oriented and put the developer and the customer in focus. As a consequence, SCM has to shift its primary focus from control activities to that of service and support activities. The main focus on audits and control needs to be replaced by a main focus on supporting the highly iterative way of working of both the team and the developers, as seen in Figure 2. From a QA point of view, the control measures are moved down to the developers themselves with the purpose of shortening the feedback loop in Agile methods. So SCM does not directly contribute to the QA on an Agile project,

this is the task of the processes that the Agile method in question prescribes. However, by supporting said processes and making them easier and safer to practice SCM indirectly is a big factor in QA on Agile projects.

The traditional process-oriented view on SCM has also lead to several misconceptions of Agile methods from an SCM point of view. The lack of explicit use of SCM and its terminology has lead quite a few people to conclude that Agile methods are not safe due to an apparent lack of rigorous change management. However, a lot of SCM activities are actually carried out in Agile methods although they are not mentioned explicitly. Bendix et al (Bendix et al, 2002) identify a need for better support from SCM, in particular for refactoring in order for this practice to be viable. Koskela (Koskela, 2003) reviews Agile methods in general from an SCM point of view and concludes that only a few of the existing Agile methods take SCM explicitly into account. He also notices that most methods highly value SCM tool support but that SCM planning has been completely forgotten. There is thus a need to provide guidance for using SCM or for implementing SCM in Agile. The SCM literature mostly takes the control oriented view of SCM (Berlack, 1992), (Buckley, 1993), (Hass, 2003), (Leon, 2005) and there is very little written about team and developer oriented support activities (Babich, 1986), (Mikkelsen et al, 1997), (Bendix et al, 2001) (Berczuk et al, 2003). These activities are the ones that can benefit Agile methods the most and should therefore be emphasized more when used in an Agile setting. However, it is important to stress that Agile methods need the whole range of SCM support from developer through to customer.

In the next section, we provide background information about SCM for those who are not so familiar with SCM, and describe and define a number of SCM-related Agile activities to establish a terminology. In section 3, we give general guidelines for how these Agile activities can be supported by SCM and how Agile methods could benefit from adopting more SCM principles. We also provide pointers to literature where more details can be found. Future trends for SCM in the Agile context are described in section 4, and in section 5 we draw our conclusions.

(2) BACKGROUND

This section gives an introduction to the concepts and terminology in SCM that serve as a background for the analysis in following sections. We also define and describe activities in Agile methods that are related to SCM or affected by SCM in one way or the other.

(2.1) SCM Activities

SCM is a method for controlling the development and modifications of software systems and products during their entire life cycle (Crnkovic et al, 2003). From this viewpoint SCM is traditionally divided into the following activities: Configuration Identification, Configuration Control, Configuration Status Accounting and Configuration Audit (Leon, 2005). These activities reflect mostly the part of a development project with relations to the customer. However, since Agile methods are often more developer centric there is also a need for a more developer oriented view of SCM than the traditional control oriented view above. Typical developer oriented aspects of SCM include: Version Control, Build Management, Workspace Management, Concurrency Control, Change Management, and Release Management (Bendix et al, 2001). We present each activity from a general perspective and explain both its purpose and what is included in the activity. After this introduction the reader should be familiar with these basic SCM concepts and their purpose, so we can use them for our analysis in section 3.

Configuration Identification
Configuration Identification is the activity where a system is divided into uniquely identifiable components, called configuration items, for the purpose of software configuration management. The physical and functional characteristics of each configuration item are documented including its interfaces and change history. Each configuration item is given a unique identifier and version to distinguish it from other items and other versions of the same item. This allows us to reason about a system in a consistent way both regarding its structure and history. Each item can be either a single unit or a collection (configuration) of lower level items allowing hierarchical composition. During Configuration Identification a project baseline and its contents are also defined, which helps to control change as all changes apply to this uniquely defined baseline.

Configuration Control
Software is very different from hardware as it can be changed quickly and easily, but doing so in an uncontrolled manner often leads to chaos. Configuration Control is about enabling this flexibility in a controlled way through formal change control procedures including the following steps: evaluation, coordination, approval or disapproval, and implementation of changes. A proposed change request typically originates from requests for new features, enhancement of existing features, bug reports, etc. A request is first evaluated by a Change Control Board (CCB) that approves or disapproves the request. An impact analysis is performed by the CCB to determine how the change would affect the system if implemented. If a request is approved, the proposed change is assigned to a developer for implementation. This implementation then needs to be verified through testing to ensure that the change has been implemented as agreed upon before the CCB can finally close the change request.

Configuration Status Accounting
Developers are able to track the current status of changes by formalizing the recording and reporting of established configuration items, status of proposed changes, and implementation of approved changes. Configuration Status Accounting is the task to provide all kinds of information related to configuration items and the activities that affect them. This also includes change logs, progress reports, and transaction logs. Configuration Status Accounting enables tracking of the complete history of a software product at any time during its life cycle and also allows changes to be tracked compared to a particular baseline.

Configuration Audits
The process of determining whether a configuration item, for instance a release, conforms to its configuration documents is called Configuration Audit. There are several kinds of audits each with its own purpose but with the common goal to ensure that development plans and processes are followed. A functional configuration audit is a formal evaluation that a configuration item has achieved the performance characteristics and functions defined in its configuration document. This process often involves testing of various kinds. A physical configuration audit determines the conformity between the actual produced configuration item and the configuration according to the configuration documents. A typical example is to ensure that all items identified during configuration identification are included in a product baseline prior to shipping. An in-process audit ensures that the defined SCM activities are being properly applied and controlled and is typically carried out by a QA team.

Version Control
A Version Control system is an invaluable tool in providing history tracking for configuration items. Items are stored, versions are created, and their historical development is registered and conveniently accessible. A fundamental invariant is that versions are immutable. This means that as soon as a configuration item is given a version number, we are assured that it is unique and its contents cannot be changed unless we create a new version. We can therefore recreate any version at any point in time. Version Control systems typically support Configuration Status Accounting by providing automatic support for history tracking of configuration items. Furthermore, changes between individual versions of a configuration item can be compared automatically and various logs are typically attached to versions of a configuration item.

Build Management
Build Management handles the problem of putting together modules in order to build a running system. The description of dependencies and information about how to compile items are given in a system model, which is used to derive object code and to link it together. Multiple variants of the same system can be described in a single system model and the Build Management tool will derive different configurations, effectively building a tailored system for each platform or product variant. The build process is most often automated, ranging from simple build scripts to compilation in heterogeneous environments with support for parallel compilation. Incremental builds, that only compile and link what has changed, can be used during development for fast turn around times, while a full build, rebuilding the entire system from scratch, is normally used during system integration and release.

Workspace Management
The different versions of configuration items in a project are usually kept in a repository by the Version Control tool. Because these versions must be immutable, developers cannot be allowed to work directly within this repository. Instead, they have to take out a copy, modify it, and add the modified copy to the repository. This also allows developers to work in a controlled environment where they are protected from other people's changes and where they can test their own changes prior to releasing them to the repository. Workspace Management must provide functionality to create a workspace from a selected set of files in the repository. At the termination of that workspace, all changes performed in the workspace need to be added to the repository. While working in the workspace, a developer needs to update his workspace, in a controlled fashion, with changes that other people may have added to the repository.

Concurrency Control
When multiple developers work on the same system at the same time, they need a way to synchronize their work. Otherwise it may happen that more than one developer make changes to the same set of files or modules. If this situation is not detected or avoided, the last developer to add his changes to the repository will effectively erase the changes made by others. The standard way to avoid this situation is to provide a locking mechanism, such that only the developer who has the lock can change the file. A more flexible solution is to allow people to work in parallel and then to provide a merge facility that can combine changes made to the same file. Compatible changes can be merged automatically while incompatible changes will result in a merge conflict that has to be resolved manually. It is worth noticing that conflicts are resolved in the workspace of the developer that triggered the conflict, who is the proper person to resolve it.

Change Management
There are multiple and complex reasons for changes, and Change Management needs to cover all types of changes to a system. Change Management includes tools and processes that support the organization and tracking of changes from the origin of the change to the approval of the 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 a change request and its actual implementation, but also to allow each piece of code to be associated to an explicit change request. Change Management is also used to provide valuable metrics about the progress of project execution.

Release Management
Release Management deals with both the formal aspects of the company releasing to the customer and the more informal aspects of the developers releasing to the project. For a customer release we need to carry out both a physical and a functional configuration audit before the actual release. In order to be able to later re-create a release, we can use a bill-of-material that records what went into the release and how it was built. Releasing changes to the project is a matter of how to integrate changes from the developers. We need to decide on when and how that is done, and in particular on the "quality" of the changes before they may be released.

(2.2) Agile Activities
This section identifies a set of Agile activities that either implement SCM activities or are directly affected by SCM activities. The presentation builds on our view of Agile methods as being incremental, co-operative, and adaptive. Incremental in that they stress continuous delivery with short release cycles. Co-operative in that they rely on teams of motivated individuals working towards a common goal. Adaptive in that they welcome changing requirements and reflect on how to become more effective. While all activities presented in this section may not be available in every Agile method, we consider them representative for the Agile way of developing software.

Parallel Work
Most projects contain some kind of Parallel Work, either by partitioning a project into sub-projects that are developed in parallel, or by implementing multiple features in parallel. Traditional projects often try to split projects into disjoint sub-projects that are later combined into a whole. The incremental and adaptive nature of Agile methods require integration to be done continuously since new features are added as their need is discovered. Agile methods will therefore inevitably lead to co-operative work on the same, shared code base, which needs to be co-ordinated. To become effective the developers need support to work on new features in isolation and then merge their features into the shared code base.

Continuous Integration
Continuous Integration means that members of a team integrate their changes frequently. This allows all developers to benefit from a change as soon as possible, and enables early testing of changes in their real context. Continuous Integration also implies that each member should integrate changes from the rest of the team for early detection of incompatible changes. The frequent integration decreases the overall integration cost since incompatible changes are detected and resolved early, in turn reducing the complex integration problems that are common in traditional projects that integrate less often.

Regular Builds
Agile projects value frequent releases of software to the customer and rapid feedback. This implies more frequent builds than in traditional projects. Releases, providing added value to the customer, need to be built regularly, perhaps on a weekly or monthly basis. Internal builds, used by the team only, have to be extremely quick to enable rapid feedback during Continuous Integration and Test-Driven Development. This requires builds to be automated to a large extent to be feasible in practice.

Refactoring
The incremental nature of Agile methods requires continuous Refactoring of code to maintain high quality. Refactorings need to be carried out as a series of steps that are reversible, so one can always back out if a Refactoring does not work. This practice relies heavily on automated testing to ensure that a change does not break the system. In practice, this also means that it requires quick builds when verifying behavioural preservation of each step.

Test-Driven Development
Test-Driven Development is the practice that tests drive the design and implementation of new features. Implementation of tests and production code is interleaved to provide rapid feedback on implementation and design decisions. Automated testing builds a foundation for many of the presented practices and requires extremely quick builds to enable a short feedback loop.

Planning Game
The Planning Game handles scheduling of an XP project. While not all Agile methods have an explicit Planning Game, they surely have some kind of lightweight iterative planning. We emphasize planning activities such as what features to implement, how to manage changes, and how to assign team resources. This kind of planning shares many characteristics with the handling of change requests in traditional projects.

(3) SCM IN AN AGILE CONTEXT

In the previous section, we defined some Agile activities that are related to SCM and we also outlined and described the activities that make up the field of SCM. In this section, we will show how SCM can provide support for such Agile activities so they succeed and also how Agile methods can gain even more value from SCM. It was demonstrated in (Asklund et al, 2004) that Agile methods, in this case exemplified by XP, do not go against the fundamental principles of SCM. However, it also showed that, in general, Agile methods do not provide neither explicit nor complete guidance for using or implementing SCM. Furthermore, the focus of SCM also needs to shift from control to service and support (Angstadt, 2000) when used in Agile. SCM does not require compliance from Agile, but has a lot of good advice that you can adapt to your project if you feel the need for it – and thus value people and interactions before tools and processes (Agile Manifesto, 2001).

In this section, we first look at how SCM can support and service the Agile activities we defined in the previous section. Next, we look at how Agile methods could add new activities and processes from SCM and in this way obtain the full benefit of support from SCM.

(3.1) How Can SCM Techniques Support Agile?

SCM is not just about control and stopping changes. It actually provides a whole range of techniques and processes that can service and support also Agile development teams. Agile

methods may tell you what you should do in order to be agile or lean, but in most cases they are also very lean in actually giving advice on how to carry out these Agile processes. In this sub-section, we show how SCM techniques can be used to support and strengthen the following SCM-related Agile activities: Parallel Work, Continuous Integration, Regular Builds, Refactoring, Test-Driven Development, and Planning Game.

Parallel Work

Agile teams will be working in parallel on the same system. Not only on different parts of the system, leading to shared data, but also on the same parts of the system, leading to simultaneous update and double maintenance. Babich (Babich, 1986) explains all the possible problems there are when co-ordinating a team working in parallel – and also the solutions.

The most common way of letting people work in parallel is not to have collective code ownership, but private code ownership and locking of files that need to be changed. This leads to a "split and combine" strategy where only one person owns some specific code and is allowed to change it. Industry believes that this solves the problem, but the "shared data" problem (Babich, 1986) shows that even this apparently safe practice has its problems, e.g., combining the splits. These problems are obviously present if you practise Parallel Work as well. In addition we have to solve the "simultaneous update" problem and the "double maintenance" problem, when people actually work on the same file(s) in parallel.

The "shared data" problem is fairly simple to solve – if the problem is sharing, then isolate yourself. Create a physical or virtual workspace that contains all of the code and use that to work in splendid isolation from other people's changes. Obviously you cannot ignore that other people make changes, but having your own workspace, you are in command of when to "take in" those changes and will be perfectly aware of what is happening.

The "simultaneous update" problem only occurs for collective code ownership where more people make changes to the same code at the same time. Again the solution is fairly simple, you must be able to detect that the latest version, commonly found in the central repository, is not the version that you used for making your changes. If that is not the case, it means that someone has worked in parallel and has put a new version into the repository. If you add your version to the repository it will "shadow" the previous version and effectively undo the changes done in that version. If you do not have versioning, the new version will simply overwrite and erase permanently the other person's changes. Instead you must "integrate" the parallel changes and put the resulting combined change into the repository or file system. There are tools that can help you in performing this merge.

The "double maintenance" problem is a consequence of the "protection" from the "shared data" problem. In the multiple workspaces we will have multiple copies of every file and according to Babich (Babich, 1986) they will soon cease to be identical. When we make a change to a file in one workspace, we will have to make the same change to the same file in all the other workspaces to keep the file identical in all copies. It sounds complicated but is really simple, even though it requires some discipline. Once you have made a change, you put it in the repository and – sooner or later – the other people will take it in from the repository and integrate it if they have made changes in parallel (see the "simultaneous update" problem).

A special case of Parallel Work is distributed development where the developers are physically separated. This situation is well known in the SCM community and the described

solutions (Bellagio et al, 2005) are equally applicable to distributed development as to Parallel Work. There are solutions that make tools scale to this setting as well. Distributed development is thus not different from Parallel Work from a SCM perspective, as long as the development process that SCM supports scales to distributed development.

In summary, we need a repository where we can store all the shared files and a workspace where we can change the files. The most important aspect of the repository is that it can detect Parallel Work and that it can help us in sorting out such Parallel Work. Also it should be easy and simple to create whole workspaces. Most Version Control tools are able to do that and there is no need to use locking, which prevents real Parallel Work, since optimistic sharing works well. We must thus choose a tool that can implement the copy-merge work model (Feiler, 1991).

Continuous Integration

In traditional projects, the integration of the contributions of many people is always a painful process that can take days or even weeks. Therefore, Continuous Integration seems like a mission impossible, but this is actually not the case. The reason why integration is painful can be found in the "double maintenance" problem (Babich, 1986) – the longer we carry on the double maintenance without integrating changes, the greater the task of integration will be. So there are good reasons for integrating as often as possible, for instance after each added feature.

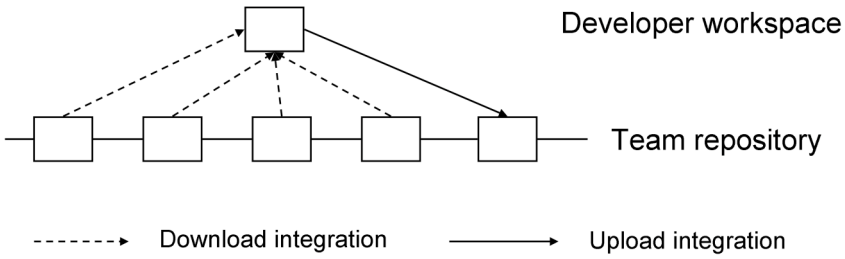

Figure 3. Download and upload integration.

Integrating your change into the team's shared repository is often a two-step process. The reason is that tools usually cannot solve merge conflicts and re-run automated tests to check the quality in one-step. First you have to carry out a "download" (or subscription) integration, where you take all the changes that have been added to the repository since you last integrated and integrate them into your workspace, as shown in Figure 3, where a box represents a new version of the configuration. If nothing new has happened you are safe and can do the "upload" (or publication) integration, which simply adds your changes as the latest versions in the repository. If something has changed in the repository, it can be either new versions of files that you have not changed – these can simply be copied into your workspace – or files that you have changed where there may be conflicting changes. In the latter case you have to merge the repository changes into your own changes. At this point, all other people's changes have been integrated with your changes and your workspace is up-to-date, so you could just add the result to the repository. However, you should check that the integration actually produced a viable result and check the quality of it. This can be done by running a set of quality tests, e.g. unit tests, acceptance tests, and if everything works well, then you can add the result to the repository – if your workspace is still up-to-date. Otherwise you have to continue to do "download" integrations and quality checks until you finally succeed and can do the "upload" integration, as shown in Figure 3.

This way of working (except for the upload quality control) is implemented in the strict long transactions work model (Feiler, 1991). You will notice that in this process the upload integration is a simple copy of a consistent and quality assured workspace. All the work is performed in the download integration. Following the advice of Babich (Babich, 1986) this burden can be lessened if it is carried out often, as the changes you have to integrate are smaller. So for your own sake you should download integrate as often as possible. Moreover, for the sake of the team you should upload (publish) immediately when you have finished a task or story so other people get the possibility to synchronize their work with yours.

What we have described here is the commonality between the slightly different models and approaches presented in (Aiello, 2003), (Appleton et al, 2003a), (Appleton et al, 2003b), (Appleton et al, 2004a), (Appleton et al, 2005), (Farah, 2004), (Fowler et al, 2006), (Moreira, 2004) and (Sayko, 2004). If you are interested in the details about how you can vary your approach to Continuous Integration depending on your context, you can consult the references.

Continuous Integration leads to an increased velocity of change compared to traditional development. This puts additional strains on the integration process but is not a performance problem on the actual integration per se. However, there may be performance issues when the integration is combined with a quality gate mechanism used to determine whether changes are of sufficient quality to be integrated in the common repository or not. Even if this quality gate process is fully automated, it will be much slower than the actual merge and upload operation and may become a bottleneck in the integration process. It may therefore not always be possible to be true to the ideal that developers should carefully test their code before uploading their changes in which case you could use a more complex model for Continuous Integration (Fowler et al, 2006) that we will describe next under Regular Builds.

Regular Builds

When releases become frequent it also becomes important to be able to build and release in a lean way. If not, much time will be “wasted” in producing these releases that are needed to get customer feedback. Making it lean can be done in three ways: having always releasable code in the repository, performing a less formal release process, and automation of the build and release processes.

Before you can even think about releasing you code, you have to assure that the code you have is of good quality. In traditional development methods this is often done by a separate team that integrates the code and does QA. In Agile, this is done by the developers as they go. The ideal situation is that the code in the repository is always of the highest quality and releasable at any time. This is not always possible and you can then use a mix between the traditional way and the Agile ideal by having multiple development lines. The developers use an integration line to check in high quality code and to stay in sync with the rest of the developers. The QA-team uses a separate line to pull in changes from the integration line and does a proper and formal QA before they “promote” the approved change to the release line, as seen in Figure 4.

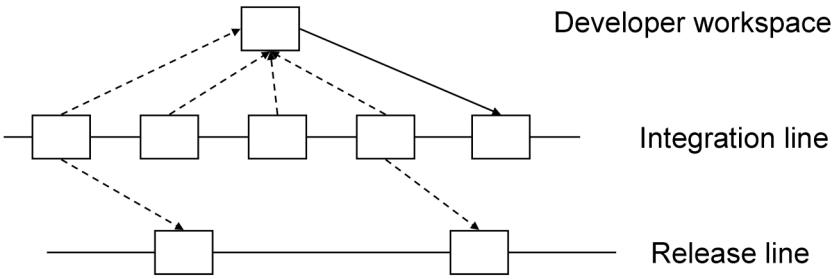

Figure 4. Working with integration and release lines.

In Agile methods there is a general tendency to move the focus of QA from coming late in the development process, just before release, to being a centre of attention as early as possible in the development process. This means that Agile can do with a less formal release process than traditional projects, because much of the work has already been done. However, there is still a need to do physical and functional audits and to work with bill-of-materials such that earlier releases can be re-created again if needed. In Agile methods, functional audits can be carried out by running the acceptance tests. They are the specification of the requirements that should be implemented. To be really sure that we have implemented everything we claim, we should check the list of acceptance tests against the list of requirements we claim have been implemented in this release. We also need to check whether all the files that should be in the release, e.g., configuration files, manual, documentation, etc., are actually there.

When releasing becomes a frequent action, there is a much greater need to automate it. The actual creation of the release can be automated by using build tools; acceptance tests and the verification that all files are there can be automated by writing simple scripts.

More information about Regular Builds can be found in (Appleton et al, 2004b).

Refactoring

Refactoring is an important part of Agile methods but also to some extent in traditional methods. The purpose of a Refactoring is not to implement new functionality, but rather to simplify the code and design.

In general there are two different situations where you do refactorings: as part of a story to simplify the code before and/or after the implementation of the story's functionality; and architectural refactorings that are needed to implement a whole new set of features. In both cases the two main problems are that a Refactoring may touch large parts of the code and that the Refactoring should be traceable and possible to undo. The latter means that there is the need for Version Control tool to keep track of the steps of each Refactoring and make it possible to back out of a Refactoring if it turns out that is does not work.

The fact that Refactorings tend to be "global", possibly affecting large parts of the code, puts even greater strains on the Continuous Integration since there are more possibilities of merge conflicts. The recommendation for successful application of Continuous Integration is to integrate very often to reduce the risk of merge conflicts. The same goes for Refactorings that should be split up into many small steps that are integrated immediately when they are done.

If you need to refactor code to facilitate the ease of implementing a story, then this Refactoring should be seen as a separate step and integrated separately – the same goes if you need to refactor after the implementation of the story. For the architectural Refactorings, we

need to split the Refactoring up into smaller tasks such that there will be as little time as possible between integrations to lower the risk of merge conflicts. Larger Refactorings should also be planned and analysed for impact such that it is possible to co-ordinate the work to keep down the parallel work, or at least to make people aware of the fact that it is going on.

For a more specific treatment of the problems architectural Refactorings can cause to SCM tools and the Continuous Integration process and how these problems can be dealt with, we refer the reader to (Ekman et al, 2004) and (Dig et al, 2006).

Test-Driven Development

The short version of Test-Driven Development is: design a little – where you design and write tests; code a little; and finally run the tests to get feedback. Here the crucial part is to get feedback on what you have just changed or added. If that cannot happen very quickly, Test-Driven Development breaks down with respect to doing it in small increments. If you want to run you tests after writing a little code, you must be able to re-build the application you want to test very quickly – if you have to wait too long you are tempted to not follow the process as it is intended.

So what is needed is extremely quick re-builds, a matter of a few minutes or less, and the good news is that SCM can provide that. There are techniques for doing minimal, incremental builds that will give you a fast turn-around time, so you can run your tests often without having to wait too long. Make (Feldman, 1979) is the ancestor of all minimal, incremental build tools, but there exists a lot of research on how to trade "consistency" of a build for time (Schwanke, 1988), (Adams et al, 1989). For the small pair development loop in Figure 2, we might be satisfied with less than 100% consistency of the build as long as it is blisteringly fast. For the big team development loop in Figure 2, i.e., integrating with others, speed might not be that important while consistency of the build is crucial. A properly set up SCM system will allow developers to have flexible build strategies that are tailored to specific parts of their development cycle.

Another aspect of Test-Driven Development is that if we get an unexpected result of a test-run, then we have to go bug hunting. What is it that has caused the malfunction? If you run tests often, it means that you introduced the bug in the code that you wrote most recently – or as Babich puts it "an ounce of derivation is worth a pound of analysis" (Babich, 1986) – meaning that if we can tell the difference in the code between now and before, we are well under way with finding the bug. Version Control tools provide functionality for showing the difference between two versions of the same file and some tools can even show the structural differences between two versions of a configuration.

Planning Game

Agile methods use stories, or similar lightweight specification techniques, as the way that customers specify the requirements of the system, and acceptance tests to specify the detailed functionality. These stories specify changes to the system and correspond to change requests when analyzed from an SCM perspective. The stories, or change requests, have to be estimated for implementation cost by the developers and then prioritised and scheduled by the customer during the Planning Game. For someone coming from SCM this sounds very much like the traditional way of handling change requests: an impact analysis has to be carried out to provide sufficient information for the Change Control Board to be able to make its decision whether to implement the change request, defer it or reject it. So we can see that the parallel to estimation in Agile is impact analysis (Bohner et al, 1996) in traditional SCM.

Likewise, the parallel to the customer prioritising is the chair of the Change Control Board taking decisions (Daniels, 1985). For the Planning Game to work properly, it is important that everyone is aware of what his or her role is – and that they seek information that will allow them to fill that role well. It is also important to be aware of the fact that the traditional formal change request handling process can indeed – and should – be scaled to fit the agility and informality that is needed in an Agile method.

(3.2) How Can SCM Add More Value To Agile?

In Agile methods there is very much focus on the developers and the production process. In the previous sub-section, we have seen how many of these processes can be well supported by techniques and principles from SCM. However, Agile methods often overlook the aspects of SCM that deal with the relation to the customer and where traditional SCM has special emphasis. In the following, we look at the traditional SCM as represented by the four activities of Configuration Identification, Configuration Control, Configuration Status Accounting and Configuration Audit (Leon, 2005). For each activity we describe what new activities and processes could be added to Agile methods to help provide a more covering support for the development team.

Configuration Identification

The most relevant part of Configuration Identification for Agile methods is the identification and organisation of configuration items. Some artefacts are so important for a project that they become configuration items and go into the shared repository. Other artefacts, e.g., sketches, experiments, notes, etc., have a more private nature and they should not be shared in order not to confuse other people. However, it may still be convenient to save and version some of the private artefacts to benefit from versioning even though they are not configuration items. They can be put into the repository but it is very important that the artefacts, configuration items and not, are structured in such a way that it is absolutely clear what is a configuration item and what is a private artefact. Structuring of the repository is an activity that is important also when it contains only configuration items.

Configuration Identification is an SCM activity that traditionally is done up-front, which goes against the Agile philosophy. However, there can be some reason in actually trying to follow the experience that SCM provides. Rules for identifying configuration items should be agreed upon, such that they can be put into the repository and information about them shared as early as possible. More importantly, though, is that the structuring of configuration items should not be allowed to just grow as the project proceeds, because most SCM tools do not support name space versioning (Milligan, 2003), i.e., handling structural changes to the repository while retaining the change history.

Configuration Control

The part of Configuration Control that deals with the handling of change requests is taken care of by a Planning Game or similar activity. However, two important aspects of Configuration Control are neglected by most Agile methods: tracking and traceability.

In traditional SCM, change requests are tracked through their entire lifetime from conception to completion. At any given point in time, it is important to know the current state of the change request and who has been assigned responsibility for it. This can benefit Agile methods too as they also need to manage changes and co-ordinate the work of different people. In some Agile methods there is an explicit Tracker role (chromatic, 2003) that is responsible for this activity.

Traceability is an important property of traditional SCM and is sometimes claimed to be the main reason for having SCM. It should be possible to trace changes made to a file back to the specific change request they implement. Likewise it should be possible to trace the files that were changed when implementing a certain change request. The files that are related to a change request are not just source code files, but all files that are affected by that change, e.g., test cases, documentation, etc. Another aspect of traceability is to be able to know exactly what went into a specific build or release – and what configurations contain a specific version of a specific file. The main advantage of having good traceability is to allow for a better impact analysis, so we can be informed of the consequences of changes and improve the co-ordination between people on the team.

Configuration Status Accounting
This activity should be seen as a service to everyone involved in a project including developers and the customer, even though it traditionally has been used primarily by management and in particular project managers. Configuration Status Accounting can be looked at as simple data mining where you collect and present data of interest. Many Agile methods are very code centred and the repository is the place where we keep the configuration items that are important for the project, so it is natural to place the meta-data to mine in the same repository. Configuration Status Accounting does not need to be an upfront activity like Configuration Identification, but can be added as you discover the need. However, you should be aware that the later you start collecting data to mine, the less data and history you get to mine. Usually this is seen as an activity that benefits only managers, but there can be much support for the developers too – all you have to do it to say what kind of meta-data you want collected and how you want it to be presented. If you do not document changes in writing, then it is important that you can get hold of the person that did a change; when you have Shared Code, then it is important to see who is currently working on what.

Configuration Audit
Configuration Audit can be looked at as a verification activity. The actual work, considered as a QA activity, has been done elsewhere as part of other processes, but during the Configuration Audits it gets verified that it has actually been carried out. The functional configuration audits verify that we have taken care of and properly closed all change requests scheduled for a specific build or release. The physical configuration audit is a “sanity check” that covers the physical aspects, e.g., that all components/files are there (CD, box, manual) and that it can actually be installed. Even though Configuration Audit is not directly a QA activity, it contributes to the quality of the product by verifying that certain SCM and QA activities have actually been carried out as agreed upon. Configuration Audits are needed not because we mistrust people, but because from time to time people can be careless and forget something. The basis for automating the functional configuration audit in Agile is there through the use of unit and acceptance tests.

SCM Plans and Roles
You definitely need to plan and design your SCM activities and processes very carefully on an Agile project. Moreover, they have to be carried out differently from how they are done on traditional projects and the developers will need to know more about SCM because they are doing more of it on an Agile project.
This does not imply that you should write big detailed SCM plans the same way as it is being done for traditional projects. The Agile Manifesto (Agile Manifesto, 2001) values working software over comprehensive documentation. The same goes for SCM, where you should value working SCM processes over comprehensive SCM plans. In general, what needs to be

documented are processes and activities that are either complex or carried out rarely. The documentation needs to be kept alive and used – otherwise it will not be up-to-date and should be discarded. We can rely on face-to-face conversation to convey information within a team when working in small groups and maybe even in pairs. However, if the team grows or we have a high turnover of personnel, that might call for more documentation. If possible, processes should be automated, in which case they are also documented.
In general, Agile projects do not have the same specialization in roles as on traditional projects. Everyone participates in all aspects of the project and should be able to carry out everything – at least in theory. This means that all developers should have sufficient knowledge about SCM to be able carry out SCM-related activities by themselves. There will, for instance, not be a dedicated SCM-person to do daily or weekly builds or releases on an Agile project. However, to do the design of the SCM-related work processes even an Agile team will need the help of an SCM expert, who should work in close collaboration with the team such that individuals and interaction are valued over processes and tools (Agile Manifesto, 2001).

SCM Tools
In general, SCM is very process centric and could, in theory, be carried out by following these processes manually. However, Agile methods try to automate the most frequently used processes and have tools take care of them, e.g., repository tools, build tools, automated merge tools, etc. Fortunately, the requirements that Agile methods have to SCM tooling is not very demanding and can, more or less easily, be satisfied by most tools. For this reason we do not want to give any tool recommendations or discuss specific tools, but rather focus on the general requirements and a couple of things to look out for. Furthermore, most often you just used the tool that is given or the selection is based on political issues.
Using Parallel Work, we would need a tool that works without locking and thus have powerful merge capabilities to get as painless an integration as possible. Using Test-Driven Development we need to build very often, so a fast build tool is very helpful – and preferably it will be flexible such that we can sometimes choose to trade speed for consistency. Working always against baselines, it would be nice if the repository tool would automatically handle bound configurations (Asklund et al, 1999) so we should not do that manually.
However, a couple of things should be taken into account about SCM tooling. Because of Refactoring and the fact that the architecture is grown organically, there will be changes to the structure of the files in the repository. This means that if the tool does not support name space versioning (Milligan, 2003), we will have a harder time because we loose history information and have no support for merging differing structures. However, this can be handled manually and by not carrying out structural changes in parallel with other work. It is much more problematic to actually change your repository tool in the middle of a project. Often you can migrate the code and the versions, but you loose the meta-data that is equally as valuable for your work as the actual code. So, if possible, you should try to anticipate the possible success and growth of the project and make sure that the tool will scale to match future requirements.

(4) FUTURE TRENDS
While resisting the temptation to predict the future, we can safely assume that the increased use and awareness of SCM in Agile development will result in a body of best practices and increased interaction between Agile and SCM activities. Furthermore, we expect to see progress in tool support, including better merge support and increased traceability to name a few.

Continuous Integration is an activity that has already received much attention and is quite mature and well understood. Many other SCM-related activities require Continuous Integration and we expect to see them mature accordingly when that foundation is now set. This will result in new best practices and perhaps specific SCM-related sub-practices to make these best practices explicit. A first attempt to specify SCM sub-practices for an Agile setting is presented in (Asklund et al, 2004) and we expect them to mature and more sub-practices to follow.

SCM tools provide invaluable support and we envision two future trends. There is a trend to integrate various SCM-related tools into suites that support the entire line of SCM activities. These tools can be configured to adhere to pretty much any desired development process. They may, however, be somewhat heavyweight for an Agile setting and as a contrast we see the use of more lightweight tools. Most SCM activities described in this chapter can be supported by simple merge tools with concurrency detection.

Parallel Work with collective code ownership can benefit from improved merge support. Current merge tools often operate on plain text at the file level and could be improved by using more fine-grained merge control, perhaps even with syntactic and partially semantics aware merge. An alternative approach is to use very fine-grained merge combined with support for increased awareness to lower the risk of merge conflicts. The increased use of SCM will also require merge support for other artefacts than source files.

The use of SCM in Agile development will enable better support for traceability and tracking of changes. A little extra effort can provide bi-directional traceability between requirements, defects, and implementation. However, more experience is needed to determine actual benefits in an Agile context, before one can motivate and justify this extra "overhead".

SCM is being used more and more in Agile methods, despite not being mentioned explicitly. However, it is often carried out in the same way as in traditional projects, but can benefit from being adapted to this new setting. The practices presented in this chapter adapt SCM for Agile methods but more widespread use will lead to even more tailored SCM. In particular, SCM practices will be further refined to fit an Agile environment and probably lead to more agile SCM. Some of these ideas may indeed transfer to traditional projects, providing more lightweight SCM in that setting as well.

(5) CONCLUSION

SCM provide valuable activities that enhance the QA for Agile development. The main quality enhancement does not stem directly from SCM but indirectly by supporting other quality enhancing activities. Traceability is for instance crucial to evaluate any kind of quality work, and configuration audits verify that SCM and QA activities have been carried out.

We have shown how typical Agile activities can be supported directly by SCM techniques while retaining their agile properties. For instance, Continuous Integration demands support from SCM tools and processes to succeed while build and release management can help to streamline the release process to enable frequent releases. SCM can thus be used to support and strengthen such developer oriented activities.

SCM is traditionally very strong in aspects that deal with the relation to the customer. Agile methods can benefit from these activities as well. Configuration Control allows precise tracking of progress and traceability for each change request. Lightweight SCM plans simplify co-ordination within a team and help in effective use of other SCM-related activities. These are areas that are often not mentioned explicitly in Agile literature.

There is in general no conflict between Agile methods and SCM – quite the contrary. Agile methods and SCM blend well together and enhance each other's strengths. Safe SCM with rigorous change management can indeed be carried out in an Agile project and be tailored to Agile requirements.

SCM tools provide help in automating many Agile activities, but we must stress that what is important are the SCM processes and not so much a particular set of tools. There are also many Agile activities that could be supported even better by enhanced tool support. For instance, current merge tools are often fairly poor at handling structural merges such as Refactorings, often this results in loss of version history and traceability, and incomprehensible merge conflicts.

Many Agile teams already benefit from SCM, but we believe that a more complete set of SCM activities can be offered to the Agile community. Tailored processes and tools will add even more value and may indeed result in SCM activities that are themselves agile which may even have an impact on more traditional software development methods.

(6) REFERENCES

Adams, R., Weinert, A., & Tichy, W. (1989). Software Change Dynamics or Half of all Ada Compilations are Redundant. In *Proceedings of the 2nd European Software Engineering Conference. Coventry, UK.*

Agile Manifesto (2001). *Manifesto for Agile Software Development*, Retrieved June 1, 2006, from http://agilemanifesto.org/

Aiello, B. (2003). Behaviorally Speaking: Continuous Integration – managing chaos for quality!. *CM Journal, September*.

Angstadt, B. (2000). SCM – More Than Support and Control. *Crosstalk – The Journal of Defence Software Engineering, March*.

Appleton, B., Berczuk, S., & Konieczka, S. (2003a). Continuous Integration – Just another buzz word?. *CM Journal, September*.

Appleton, B., Berczuk, S., & Konieczka, S. (2003b). Codeline Merging and Locking: Continuous Updates and Two-Phased Commits. *CM Journal, November.*

Appleton, B., Berczuk, S., & Konieczka, S. (2004a). Continuous Staging: Scaling Continuous Integration to Multiple Component Teams. *CM Journal, March*.

Appleton, B., & Cowham, R. (2004b). Release Management – Making it Lean and Agile. *CM Journal, August*.

Appleton, B., Berczuk, S., & Cowham, R. (2005). Branching and Merging – An Agile Perspective. *CM Journal, July*.

Asklund, U., Bendix L., Christensen H. B., & Magnusson, B. (1999). The Unified Extensional Versioning Model. In *Proceedings of the 9th International Symposium on System Configuration Management, Toulouse, France, September 5-7*.

Asklund, U., Bendix, L., & Ekman, T. (2004). Software Configuration Management Practices for eXtreme Programming Teams. In *Proceedings of the 11th Nordic Workshop on*

Programming and Software Development Tools and Techniques, Turku, Finland, August 17-19.

Babich, W. A. (1986). *Software Configuration Management – Coordination for Team Productivity*. Addison-Wesley.

Beck, K. (1999a). Embracing Change with Extreme Programming. *IEEE Computer, 32(10),* 70-77.

Beck, K. (1999b). *Extreme Programming Explained – Embrace Change*. Addison-Wesley.

Bellagio, D. E., & Milligan, T. J. (2005). *Software Configuration Management Strategies and IBM Rational ClearCase*. IBM Press.

Bendix, L., & Vinter, O. (2001). Configuration Management from a Developer's Perspective. In *Proceedings of the EuroSTAR 2001 Conference, Stockholm, Sweden, November 19-23*.

Bendix, L., & Hedin, G. (2002). Summary of the Subworkshop on Extreme Programming. *Nordic Journal of Computing, 9(3)*, 261-266.

Berczuk, S., & Appleton, S. (2003). *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley.

Berlack, H. R. (1992). *Software Configuration Management*. John Wiley & Sons.

Bohner, S. A., & Arnold, R. S. (Ed.). (1996). *Software Change Impact Analysis*. IEEE Computer Society Press.

Buckley, F. J. (1993). *Implementing Configuration Management: Hardware, Software, and Firmware*. IEEE Computer Society Press.

Chromatic, (2003). *Chromatic: Extreme Programming Pocket Guide*. O'Reilly & Associates.

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

Daniels, M. A. (1985). *Principles of Configuration Management*. Advanced Applications Consultants, Inc.

Dig, D., Nguyen, T. N., & Johnson, R. (2006). *Refactoring-aware Software Configuration Management* (Tech. Rep. UIUCDCS-R-2006-2710). Department of Computer Science, University of Illinois at Urbana-Champaign.

Ekman, T., & Asklund, U. (2004). Refactoring-aware versioning in Eclipse. *Electronic Notes in Theoretical Computer Science, 107*, 57-69.

Farah, J. (2004). Making Incremental Integration Work for You. *CM Journal, November*.

Feiler, P. H. (1991). *Configuration Management Models in Commercial Environments* (Tech.. Rep. CMU/SEI-91-TR-7). Carnegie-Mellon University/Software Engineering Institute.

Feldman, S. I. (1979). Make - A Program for Maintaining Computer Programs. *Software - Practice and Experience, 9(3)*, 255-265.

Fowler, M., & Foemmel, M. (2006). *Continuous Integration*. Retrieved June 1st, 2006, from http://www.martinfowler.com/articles/continuousIntegration.html.

Hass, A. M. (2003). *Configuration Management Principles and Practice*. Addison-Wesley.

Koskela, J. (2003). *Software configuration management in Agile methods.* VTT publications: 514, VTT Tietopalvelu.

Leon, A. (2005). *Software Configuration Management Handbook*, Artech House.

Mikkelsen, T., & Pherigo, S. (1997). *Practical Software Configuration Management: The Latenight Developer's Handbook*, Prentice Hall.

Milligan, T. (2003). *Better Software Configuration Management Means Better Business: The Seven Keys to Improving Business Value*, IBM Rational white paper.

Moreira, M. (2004). Approaching Continuous Integration. *CM Journal, November*.

Sayko, M. (2004). The Role of Incremental Integration in a Parallel Development Environment. *CM Journal, November*.

Schwanke, R. W., & Kaiser, G. E. (1988). Living With Inconsistency in Large Systems. In *Proceedings of the International Workshop on Software Version and Configuration Control, Grassau, Germany, January 27-29*.