A short introduction to Software Configuration Management

Lars Bendix

In this lecture note, I want to introduce you to the marvellous, yet little known, world of Software Configuration Management (SCM). Other Software Engineering disciplines like requirements engineering, analysis and design, programming, and testing are commonly taught at universities. Software Configuration Management, on the other hand, is largely ignored at universities – yet very appreciated in companies. Actually, SCM is the topic in software engineering with the largest gap between what is taught at university and what is needed in industry [Garuosi et al., 19]. This lecture note will try to make this gap a little smaller.

Who should be interested in SCM? Anyone who will take part in the development and production of software products. In fact, for parts of SCM (those strictly related to Configuration Management) it really doesn't matter whether you develop roller skates, cars, smartphones or software – the concepts and principles are completely identical, though there may be different ways of implementing them. Other parts (still no software) are not particularly related to handling code but are useful if you handle any kind of digital information that can easily be copied – and have to develop that in collaboration with other people working in parallel on the same digital artefact. A rather small part of SCM is strictly related to developing software (and hence the S in SCM), where code has to be compiled, linked, loaded and packaged to become applications. But then again – the concepts and principles would be applicable to any kind of "production process" that could be automated.

SCM is development method agnostic – it doesn't really matter whether you follow a Waterfall-like or an Agile-like or any other development method. In any case, you have a group of people that have to work together to create a product. Whether they have different roles or are cross-functional, work still needs to move as effectively and efficiently as possible from an idea of a new functionality (a feature or a bug-fix) to a finished product that can be turned over to the customer for use.

SCM is a highly interdisciplinary field which is what makes it so interesting for people who do not want to do the same thing day in, day out. It touches most, if not all, subjects from software engineering in general and programming in particular, but it also has aspects from Collaborative Work, Working In Groups, Knowledge Management, Cognitive Science and more. This is also what makes being a configuration manager a challenging task.

SCM is probably one of the most important and cross-cutting topics in software engineering. In part, it provides the infrastructure for a software organization that allows "things" to flow around in an orderly and controlled fashion. So, no matter whether you will work with requirements, test, programming, quality assurance or project management you will get into contact with and get help from SCM. You will be a better and more appreciated employee if you know why you have to follow some apparently stupid processes and why there are some things that you should never do. In part, SCM provides a collaboration framework for a software team. It helps people coordinate their efforts and integrate contributions into coherent products. Knowing the techniques of SCM will allow you to implement them on your next project (if not already done) and make your (and your team's) life easier – and make you a valuable participant on any (software) project.

In the next chapter, I will give a general introduction to Software Configuration Management. I will explain the motivations and purpose of using SCM and give a brief history of how SCM started and has developed. Chapter 2 will cover the aspects of SCM that are important for a team in its daily work, whereas chapter 3 will cover SCM seen from the company's perspective and what is important when dealing with products and customers.

1. Introduction

In this chapter I will motivate the use of Software Configuration Management. What are the problems it was "invented" to deal with, what are the benefits from using SCM – and some sort of first definition of what SCM is. I will also give a short historic account of how (and why) *Software* Configuration Management was "born" and how it has evolved since then.

1.1 Motivation and purpose

Software projects are not the only "projects" with failure of coordination. Have you ever watched MasterChef? Then you have probably seen how some of the participants are superb at preparing a single dish by themselves. However, when they are put in charge of a team that has to produce a three-course meal – or a single meal for 100 people – they break down and everything falls apart. That is the time when Gordon Ramsay digs deep into the "interesting" part of his English vocabulary. The fish was prepared too early, so now when it has to be served it is cold. The boiled vegetables were not ready at the same time as the steak. And "someone" forgot an important part of the dessert. Gordon Ramsay is not only looking for an able cook, he is more interested in getting a person that is able to work together with all the other people in his restaurant kitchen – and someone who is able to coordinate all this so the kitchen becomes a finely tuned "delivery machine".

If you have ever played in a band (I have), you will know that not everyone is able to keep the rhythm. Sometimes it is on purpose for artistic impression, other times it is by mistake. In either case, the person needs to get in sync with the rest of the band. If everyone tries to adapt to the rhythm of each other it ends up in chaos with everyone trying to catch up with everyone else. Usually it is the drummer who is the stable person and he will ignore any deviation from the rhythm from other band members and just stick to his rhythm – much like a "drum machine". Then it is the responsibility of who gets out of rhythm to find his way back into sync. Not being an able drummer, in my band it was the bass player who was rock steady and I – and everyone else – was following him. The rhythm can be considered the processes that you have to follow on a software project – and if you for some reason deviate from them they will not change (otherwise everyone else would become confused), but you will have to find your way back into sync.

On software projects we need people who can do as they are told (cooks) and people who can orchestrate the collaboration on this team (MasterChefs). The configuration manager in a certain sense is the MasterChef – even if he might not be able to cook, he can manage and control a team of cooks, so they constantly deliver the most beautiful dishes, perfectly cooked and right on time (and budget). We also need people who can keep their "rhythm" and who knows when (and when not) they can deviate and how to get back into sync with the rest afterwards.

Imagine that there was no traffic code. How would traffic be then? Pretty chaotic! It would work if you were the only person in the world, but you are seldom alone on a software project and even when you are, we need rules or guidelines for how we can optimize the flow of work without too many accidents. You can also ask people from the Swedish outback what happens when you encounter an obstacle (reindeer or elk) that doesn't know about and respect the traffic code. We need "emergency habits" so we panic in an orderly and controlled way. You could ask an Italian – they know that there are two sets of traffic codes: the formal and the real one. Under the old traffic code the traffic in the roundabouts would break down and come to a standstill two hours earlier when the traffic police was present than if they were absent. The Italians knew that the traffic code for roundabouts had to be "interpreted", but also that there were other parts of the traffic code that should be followed by the letter. You need to know the rules and the context if you want to succeed with "improvising".

The IEEE Standard 828-2012 [IEEE828, 12] states the purpose of SCM as to:

- Identify and document the functional and physical characteristics of any product, component, result, or service
- Control any changes to such characteristics
- Record and report each change and its implementation status
- Support the audit of the products, results, services, or components to verify conformance to requirements

In more plain words this means that we need to know *what* it is that we have to develop (and document it), then we must *handle changes* to that in some way, we should keep track of how things *progress* and finally we have to *check* that what we have developed is also what we set out to develop.

In his book on SCM, Wayne Babich writes in the preface "Sometimes it is embarrassing to be a computer programmer. What other profession has such a remarkable rate of schedule and cost overrun and outright failure? … Our failures are not of the individual contributors; most of us design, code and debug adequately or even well. Rather, the failure is one of coordination. Somehow we lack the ability to take 20 or 30 good programmers and meld them into a consistently productive team. The purpose of this book is to help solve that problem." [Babich, 86] – and part of the purpose of SCM is to help with that problem.

This means that the standard takes a more top-down view of the development of the product – how it is defined and what is delivered after taking into account requested changes. This is what serves the project/company not to lose its face in front of the customer/market when a product is delivered – and further developed/maintained. Babich, on the other hand has a more bottom-up view of things – what a team needs *during* the development of the product. This serves to remove some of the confusion when a smaller or larger group of people have to collaborate to create a common product.

The IEEE 828 standard [IEEE828, 12] defines SCM as: "A discipline applying technical and administrative directions and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements".

In his book, Wayne Babich gives this definition of SCM: "I use the term in a more expansive sense. I include not just the formal release of software to the customer, but the day-to-day and minute-by-minute evolution of the software inside the development team. Controlled evolution means that you not only understand what you have when you are delivering it, but you also understand what you have while you are developing it. Control helps to obtain maximum productivity with minimal confusion when a group of programmers is working together on a common piece of software.” [Babich, 86].

As we have seen, these differing definitions give different perspectives on what SCM is and how it should be defined. Further on in this lecture note, we will see other “definitions” of SCM – each of them even if not 100% correct and covering is neither 100% wrong and will add yet another aspect to our understanding of why we are doing SCM and what it is.

1.2 History of SCM

If we drop the S from SCM then configuration management probably goes so far back in history that it cannot be traced. You could say that the Egyptians would have had to use CM principles for building the pyramids as a team effort. In particular with the industrial revolution and mass production it became evident that methods and techniques were needed to be able to manage and control the production process to guarantee a more consistent quality. A single carpenter building a table all by himself and with all the time and resources in the world would not care about configuration management. However, a team of carpenters building several similar houses under time and budget pressure would certainly use configuration management concepts and principles to succeed. So configuration management is well-known in other industries and has been practiced there for centuries with the software industry often being the exception from the rule.

In the sixties, the S was added to CM by the American Department of Defence (DoD). They, as *customers* of software products, experienced a complete mess and in particular a disturbing unpredictability in companies’ capability to deliver the required product on time and within budget. Since the software companies did not seem to want to (or be able to) do something about the situation the DoD decided to do something about it. They were used to using Configuration Management when they ordered aircraft carriers or fighter planes, so it was nothing new to them. It helped them get the product that they had specified in their order – and to handle all changes along the way to final delivery of the product. Sometimes it would turn out that a feature would be impossible to implement and they would waive not to have it – and that change would be recorded in the contract. Sometimes it would turn out that another feature would become too costly to implement and they would accept a deviation from the original “luxury” component – and that change would be recorded in the contract. Through the Configuration Management procedures they could also follow the progress of the implementation of all requirements and changes. And, finally, they “forced” the companies to check their products before they sent them to their customers. All sensible things that any customer would have expected a company to already exercise – but the software industry it different. So the Department of Defence decided to take out an old copy of their Configuration Management standard that served so well in hardware contexts – sprinkle a little “software” here and there – and add the word “Software” in front of Configuration Management. Work done – and they were happy.

Also in the sixties, other people, notably programmers, found growing problems in coordinating their parallel efforts and being programmers they created tools that could help them handle and manage their problems. From the beginning versioning was just some sort of back-up, having a central secure shared copy of everything instead of everyone having to keep their own private copies. This way was also much nicer to disk space, which was more expensive back then – but the most important gain was that everyone could know what version they were having. This gave rise to the first early version control tools like SCCS. Then developers discovered that this version control tool could be used for more. They started to abandon the focus on single files and moved to working on whole configurations with CVS as one of the first to do that. They also added more functionality as the developers found they needed something to make their life easier. This was the case for both open source tools like Subversion, git and Mercurial and for commercial tools like Synergy, ClearCase, BitKeeper and Perforce. Today version control tools have more than sufficient functionality to server most project teams and it really doesn't matter which one you chose, but more the way in which you use it – unless you have very exotic needs.

2. SCM for the team

From the historic overview (section 1.2), we saw that there have been two different driving forces behind the development and "definition" of SCM. One top-down driven by companies and customers and their needs – and another bottom-up driven by the developers and their needs. We will start with the bottom-up part since that is closest to and most useful for what you will do right after university – unless you become a configuration manager, project manager, company owner or customer (in which case you could skip to chapter 3).

Working as a developer you will collaborate with other people to create a common product. In this chapter we will assume that you are programmers working on a piece of software. However, in reality you could be a group of testers working on the test suite, a group of requirements engineers working on the requirements specification – or a group of high-school students using Dropbox or mailing around different versions of their chemistry lab report.

Collaborating with other people will be an important part of your daily work, so you really need to understand what are the potential collaboration problems so you are able to spot them when – or before – they occur and so you have an idea of how you should deal with them. This chapter is based in part on Wayne Babich' book on Software Configuration Management for Team Coordination [Babich, 86] and in part on Bendix and Vinter's paper on Software Configuration Management for Developers [Bendix et al., 01].

In the following, we will first look at the basic coordination problems and some simple preliminary ways of addressing them. Then we will look at more general and advanced coordination strategies that can be implemented with the help of most version control tool.

2.1 Coordination problems

It is an exciting day when all the different parts for an Airbus 320 arrive in Toulouse for the first time and have to be assembled. In this case the costs of coordination problems (and subsequent fixing) are very high and extreme care has been taken to avoid them. In the software industry, on the other hand, things are so easy to change (a typo in a line of code) that much less care is exercised. However, the consequences of coordination problems may be just as costly.

Some people believe that using a version control tool will solve all coordination problems – this is not so. In part it will help you, but mostly it depends on how you use the tool. So these coordination problems are never going to disappear – the best we can hope for is to be able to spot them and to manage them.

Shared Data

Now, let us go back in time to when people used a shared file server for collaboration. In some places and contexts people still do that for collaboration – whether the shared file server is called Dropbox (or similar) or using things that are outside of their control.

John, George and Paul are collaborating on an exciting new programming project. They have decided to put all the source code files on a shared file server. This way they can access them whenever they need and from wherever they may be and everyone has access to everything – and everything is nicely collected in just one place instead of being spread out. What more could they ask for? Let us see ;-)

One evening John is feeling in the mood for putting in some extra work on their project. He works until late and everything proceeds smoothly until suddenly the program crashes. Not only does it crash, but there is no apparent explanation for why it should crash. John is annoyed and blames himself for working late hours as that apparently causes him to make stupid mistakes. He sets out on the hunt for the bug and several hours later – in the early morning hours – he finally discovers what has happened. Unknowingly to John, George coming back from a party decided that now was the time to fix the bug that had bothered everyone for weeks – the square-root function consistently returning a value that was 0.1 off. However, all the parts of the program – including the new parts John programmed – knew that and corrected the value returned by the square-root function so it would become correct. Obviously George knew that and had also corrected all the places where the square-root function was used – or at least all the places that he knew of. Since he didn't know that John was also working on the program during the night he didn't know of the new places in which the function was used – and John did not know of the new behaviour of the function he was using.

Now, what was in play here? John and George are sharing the square-root function – George is fixing it and John is using it. So changes made by George will affect John. Usually that is not a problem since everyone works in the same room and communicate to make sure that everyone knows what everyone else is doing. Actually, from time to time it *is* a problem – Paul often forgets to inform the others of what he is doing. When more people are sharing something – in this case source code – changes made by one person will interfere with the work of another person. Murphy's law says that this will happen either at a time where the person doesn't know about the change or at a time where the interference will be disturbing. In the best of cases – as with John – the program will crash immediately and we will probably know where to start looking for the cause. However, in the more nasty cases it is a subtle defect and it will not show its effect until several weeks later and will make debugging very hard.

The root cause of the "Shared Data" problem is that several people are accessing and modifying the same data. That data might be source code. In that situation John completely looses control over when George' changes will hit him. As Wayne Babich put it: "there will be times where changes introduced by one programmer are an unpleasant surprise to the others" [Babich, 86]. If you are sharing something you also have the "Shared Data" problem – period. But if you know that and know how to deal with it, it doesn't have to become a problem.

Now, what should they do? Paul, not having worked all night, has a brilliant idea. Instead of the shared file server each programmer should have a local copy of all the files. That way George' changes would not have affected John and he could have worked in splendid isolation. They decide to call the place where they put the local copy for a *workspace* and everyone is happy. John and George love the idea of splendid isolation – in particular because Paul often forgets to inform about changes and sometimes make changes that do not work. Let us see how that works out.

Double Maintenance

We have identified the "Shared Data" problem – and invented workspaces as a way to deal with that problem. So now everything should work perfectly – and it does – for a while. John, George and Paul are programming away in splendid isolation and no-one interferes with the work of the others.

However, after some time they start to experience some inconveniences – and to long for the good old days with the shared file server. The problem is that Paul discovers a bug in the program and quickly fixes the code to remove the bug – but the bug is also in the copies that John and George have. Since Paul doesn't communicate that much, it takes some time before John discovers that the bug is also in his copy of the code – and the same does George when he is asked by John. When they tell Paul about the bug he answers: "I know, I fixed it hours ago". After a short argument, they ask Paul about the fix and he guarantees them that it works and is tested – so they would like to have the fix too, but how? John decides to invite Paul over to his computer to fix the bug also in his workspace. George thinks that this is pretty cool and decides to do the same. Paul is now grateful that they are only three people on the project. He also fears that he might not be able to fix the bug in exactly the same way in John's copy and in George' copy as in his own – as hard as Paul may try there is no way that he will be able to do that in the long run. So what started out as identical copies of the code slowly diverge because they all make changes to their copies – new features and new bug-fixes – and even if they try to maintain the copies equal they will most probably not be able to do it. Now, the million-dollar question is how they can get everything back together again?

The root cause of the "Double Maintenance" (or rather the "multiple maintenance") problem is that we create copies. We could stop doing that, but then we are back to the shared file server and the "Shared Data" problem. So that is not an option – we want to keep the workspaces because they serve a purpose. So when we cannot eliminate the "Double Maintenance" problem, we will have to see if we somehow can turn it into a manageable problem. A way that is a little nicer to Paul than having him repeating (or trying to) the bug-fix also in John's and George' workspaces. Paul, being scared stiff by the amount of work that might lay ahead of him, has a proposal. In the future, he will become more disciplined and immediately when he has fixed a bug, he will upload (write back) the changed source code files to the shared file server and notify the others so they can download the changed files and thus get the bug-fix into their workspace. John and George think that it sounds like something that could work – and Paul is relieved that he doesn't have to go fix things in other people's workspaces anymore.

We have now brought back the shared file server. However, since we have the workspaces the file server is not the place where we actively work (we do that in the splendidly isolated workspaces). It is a shared place that allows us to exchange changes with the purpose of easily managing the "Double Maintenance" problem by allowing people to synchronize with other people's changes. The shared file server becomes the single source of truth for our project, and is effectively turned into a shared *repository*.

Simultaneous Update

Now we have fixed the "Shared Data" problem inventing *workspaces* – and fixed the "Double Maintenance" problem the workspaces created by writing changes back to the shared *repository*. That should make life easier for our three guys. Let us see how it works out.

John, George and Paul are convinced that now their coordination problems have been dealt with and that they can finally focus on getting some coding work done. Without the coordination problems they are very productive and produce new features and new bug-fixes that they send to their customers from time to time. John and Paul are very creative and come up with new features that can provide value for their customers – George, being more meticulous – "cleans up" after the others as their features do not always work perfectly. Everyone is happy until a very angry customer calls them up one day. A bug that he reported two months ago is still there and causing him problems – even though they clearly told him that they would fix it right away and in a follow-up call a week later told him that now it had been fixed. After this rather embarrassing telephone conversation, John and Paul turn to George. A very nervous George utters: "But I did fix it, it worked and I put it up on the repository for everyone to use!" – "Prove it", John and Paul very angrily reply. They try to see if they can reproduce the bug from the code base in the repository – and there it is. George is absolutely sure that he fixed the bug and uploaded the fix. However, unfortunately for George there is no trace whatsoever of his fixed source code file in the repository – and the copy in his workspace doesn't count as proof that he actually uploaded the bug-fix, just that he fixed it locally.

What on Earth happened? Is George lying, is he sloppy or did he just forget to upload the bug-fix – or are there still some coordination problems left? Digging deeper into what might have happened, they discover that the file that fixes the bug in George' workspace has a time-stamp that is two days older than the corresponding file in the repository. Furthermore, the latest changes to the file in the repository is part of some changes that John did to implement a new feature. Giving George the benefit of the doubt they come to the conclusion that George actually fixed the bug and uploaded the fix – however, in the meantime John was working on the same file as part of his feature – and when John finally uploaded his changes they over-wrote the changes that George had uploaded – and the bug (that was in the original code John copied to his workspace) came back to life. This is called the "Simultaneous Update" problem – that when two persons are working simultaneously with the same file there is the risk that one person's changes become overwritten by the other person's changes and thus disappear. Actually they don't have to work in parallel – it is sufficient that John copied the files from the repository one second before George decided that his testing was done and he was ready to upload. Now, how do they deal with this problem?

The next morning George comes back with a solution: *versioning*. Instead of over-writing the copy in the repository every time a change is uploaded (which is what created the problem) they will add a new version of the file to the repository. George is happy – this way he will be able to point to version 7 and say: "Here, you see? I fixed the bug and it works perfectly – and I uploaded the fix – proof done!" John and Paul tend to agree with George – it does seem to fix the problem with over-writing changes. However, after some more analysis they realise that it mainly fixed George' problem about proving that he did the bug-fix, but the customer's problem that the bug comes to life again like a zombie will still be there. John will still be able to upload his new feature as version 8 – and it might not include George' bug-fix. Thus new uploads will not over-write older uploads, but they will still be able to "shadow" for older changes if they have not been integrated.

This is basically how the early very rudimentary versioning functionality came into being. Even if it is helpful in managing the three coordination problems, we have also seen that it isn't sufficient. But let us leave our three brave guys for now. We have learned that in any collaboration there will be coordination problems. We have also learned about three common coordination problems so we will be able to spot them whenever and wherever they should occur. And finally, we have learned some of the things that can help us manage these three coordination problems. Seeing that version control functionality was helpful – but not sufficient – for managing coordination problems, developers soon added more useful functionality to version control. So – we will stop here and move on to "real" version control in the next section. We will see if we can put the final nail in the coffin of the "Simultaneous Update" problem – and look at the more advanced coordination strategies that most version control tools allow us to implement.

2.2 Coordination strategies

We saw in the previous section that simple versioning functionality can be a great help in trying to manage the three coordination problems: "Shared Data", "Double Maintenance", and "Simultaneous Update". However, it was not entirely satisfactory – there were some "open issues" about the "Simultaneous Update" problem – and there may also be some inconveniences left from the "Double Maintenance" problem and the "Shared Data" problem.

In this section, we will see if we can finally fix the "Simultaneous Update" problem now that we are so close. We will also return to the other two coordination problems and see that we need more powerful and flexible ways of dealing with them in not only effective way but also efficient ways. Finally there are some additional coordination issues that are also addressed by versioning functionality.

Copying and adding files

After creating the concepts and terms workspace, repository and versioning, the next step was to establish a more "version-like" terminology in general. Copying and adding files or downloading and uploading files was too reminiscent of the shared file server that we wanted to get away from. Instead the established term for "copying files from the shared file server into the private workspace" was called a *check-out*, and the term for "adding the file from the private workspace as a new version in the repository" was called a *check-in* [Feiler, 91]. In the early version control tools you only checked out the files that you wanted to change (one by one) – and you had to keep track of which files you actually changed so you could check them in (one by one) at the end. In more modern version control tools, you just check out the whole system (so you have everything in your workspace) and the version control tool keeps track of what has been changed and should be part of the final check in (files that have not been changed should not give rise to a new version in the repository).

The field of software engineering seems to be very "standardization averse" so many version control tools have their own terminology for check-out and check-in (and other things/functionality). Synonyms for check-out are: charge-out, update, pull, get –

synonyms for check-in are: charge-in, commit, push, submit. In this lecture note we will try to consistently use *update* and *commit*.

Concurrency check

In the previous section we almost nailed the "Simultaneous Update" problem when George "invented" versioning in the repository. However, the problem that remained was that John was allowed to upload his new feature (based on version 6) as version 8 even if George had already uploaded his bug-fix (also based on version 6) as version 7. That effectively made George' bug-fix disappear – who cares about older versions, the latest and greatest rules ;-)

We need to fix that. We need a gate-keeper that will discover that George and John have been working in parallel on the same file and stop John from uploading after George has uploaded. Something that can say: "Sorry pal, you are no longer up to date – so you cannot upload – please bring yourself up to date and try again". This functionality is the concurrency check in version control tools. This functionality is so basic that without it, a tool should not be considered a version control tool. This means that the version control tool must keep track of what is the latest version in the repository – that is the easy part. However, it must also keep track of which version(s) John and George copy (download) from the repository to their respective workspaces. With this information, the tool can check if the version number of the latest version in the repository is equal the version number of the "working version" that was copied to the workspace and modified. If that is the case, the modified copy in the workspace can be added to the repository as the new latest version. If not, something new has been added and if the upload is accepted we will create the "Simultaneous Update" problem. In the latter case, the person will have to bring himself up to date with the latest version by integrating the changes from the repository into the modified version he has in his workspace. We will see later (Integration/merging) how that integration can be done.

Locking

Do you remember the situation where the concurrency check tells John that he is not up to date and that he should fix that? John is thinking hard of an easy way to integrate George' changes into his workspace so he can become up to date. He considers begging George to come over to his workspace and repeat the change and hope that they could make it work together with his changes – but as we have seen previously (the "Double Maintenance" problem) that is not of the real world. John, who is a sucker for easy and simple solutions, then invents the concept of *locking*. When he wants to change a file he simply locks the file and excludes everyone else from changing the file. You can still update and get the file into your workspace and you can also (in many cases) change the file in your workspace, but when you want to commit your changes, the version control tool will check whether you are the locker or not. If you are, then no problem and your commit will go through – if not, your commit will be aborted and you will have to come back later when the file has been unlocked. This means that people can still have and use locked files (in read mode) to build whole executables they can test, but when it comes to changing files it effectively forces people to work sequentially instead of in parallel. This is an easy fix to the problem and since there are many like John out there, most version control tools allow for locking.

Long transactions
When we make changes to a system it is rarely only to a single file. Usually a change spans several files – in particular for changes that are new features, but in many cases also for bug-fixes. Back in the good old days when all the updating and committing was done manually, it happened that you would miss committing a file that had been changed. Missing one of the parts of your change the feature – or bug-fix – would not work. Or at least it would not work in the repository (that everyone uses as “the truth”), but it would work in your workspace. That is a good start for “heated arguments” ;-)

This unfortunate “situation” led to the concept of *long transactions* [Feiler, 91]. The concept is borrowed from database and means that either all parts of your change go through in the commit or the commit is completely aborted and nothing goes through. This is also known as “atomic commits” even if that term sometimes has other connotations. The idea is that we let the version control tool keep track of what has been changed and at the time of the commit, it tries to commit all the changed files. If there is even one single file where the concurrency check fails, the entire commit is aborted. The idea is that we want to work in “logical units of change” and keep each such unit together. Most – if not all – version control tools support the concept. However, most tools also support that you can pick and choose yourself what to commit – and it happens that people think that they are more clever than the tool. So when the tool complains that “foo.bar” is not up to date they simply pick all the other changed files and commit them – don’t do that – don’t break the long transactions. You are not more clever than the tool – unless you are Linus Torvalds.

Copy/merge
Now, let us return to the “Simultaneous Update” problem. We saw earlier that John fancied the easy fix of locking files to exclude others from working in parallel with him. That worked for a while until John again and again experienced that a file he needed to change to fix a bug or implement a feature had been locked by someone else. Sometimes he was waiting for other people to unlock a file or two he needed – only to experience that they were waiting for a file or two that he had locked to be able to complete their task. He even once experienced that someone had gone on vacation without unlocking some files.

So obviously locking also has its drawbacks. John would like to have also a more flexible way of working – but still without suffering from the coordination problems. The reason that John came up with locking in the first place was that he couldn’t convince George to come over to his workspace and implement the bug-fix once more, so John’s workspace would have “integrated” George’ change and become up to date so he would be able to commit. But what if we could have a tool that wouldn’t mind coming over and fix the bug once more in John’s workspace – and Paul’s workspace – and everyone else’s workspace? And a tool that would do it the exact same way each and every time (remember the “Double Maintenance” problem?)? A tool that would never become overloaded with work, would never become bored from repetitive work, and would never loose its attention to detail. When programmers need a tool, they build one – that is the cool part of being a software engineer. So they built a tool that given a common ancestor (version 6 that both John and George started from) would integrate the changes committed by George in version 7 with the changes (from version 6) that John had in his workspace – and place the result (that could then be committed) in John’s workspace. A tool that would merge two parallel lines of changes that originated from the same ancestor – and they called it a *merge tool*. Most version control tools have a built-in merge functionality, but many allow you to use also external third-party merge tools if you are not happy with the one built in. In most cases the merge tool will have no problem in integrating two changes into one – if George modified some lines in the beginning of the file for the bug-fix and John adds a couple of methods at the end of the file for the new feature, then there is no doubt what should be the merged result. However, in some cases both George and John may have changed the same lines – and in this case the merge tool will not "randomly pick one of the changes" but simply give up and signal that there is a conflict and that some human (George or John?) intervention will be needed to solve the conflict.

With the copy-merge strategy the three lads are happy that they one more coordination strategy in their tool-box.

Long transactions revisited

Now things really start to take off and the coordination works like a dream. Or so they thought. In spite of workspace, repository, versioning, concurrency control, locking, long transactions and copy-merge, things still turn into a nightmare from time to time. John meticulously test everything and does not commit until it works – Paul does the same – the concurrency check makes sure that they have not worked in parallel on the same files. Still from time to time things stop working in the repository. After a fair amount of analysis they start to realise why that might cause the problem.

It is not John's fault – his feature is carefully tested and it works. It is not Paul's fault – his bug-fix is carefully tested and it works. At the root of the problem is the way the concurrency check mechanism works. John has worked on the set of files: A, B, C and D – Paul has worked on the files X and Y. Since the concurrency check works on single files it makes a terrible mistake. When Paul wants to commit (after John has committed) it only checks whether or not Paul is up to date with respect to the files X and Y that he has changed – and he is – and the commit goes through. However, it turns out that Paul's changes to files X and Y do not work that well with John's changes to files A, B, C and D – and when that is discovered it is too late – the problem is already in the repository from where it will contaminate everyone as they update. Effectively what is done is a (structural) merge into the repository – and, as we saw above, merges are not always perfect. When we talk about merges on files we can have physical conflicts – when it is a structural merge on systems we can have logical conflicts. In any case we would like the conflicts (the result of the merge) to end up in a workspace, where it can be contained, fixed and tested before it is finally committed.

So they are looking for a concurrency check mechanism that does not just work on single files, but on the whole system – if something somewhere is changed you are no longer up to date. And with that they get a coordination strategy that is stricter than long transactions – and they call it *strict long transactions*.

Locking revisited

With their wonderful merge tool – and the (strict) long transactions – they quickly abandon the rigid locking mechanism and go all in on the copy-merge strategy. For a while it works perfectly and everyone is happy for the flexibility – but one day John starts to long for the good old days of the "quick fix" – that is the locking.

What happens is that they discover that providing a software product is more than just lines of code and test cases. They also need to produce a user’s manual. They decide to write that in Microsoft Word (don’t ask why) – and a user’s manual is not worth anything if it doesn’t contain a lot of screen-dumps. So now it is not just files with lines of text they have to work and coordinate on – they also have to handle .docx files and .jpg files. One fine day when John has to update the user’s manual after implementing a new feature he discovers that he is not up to date – Paul has been working in parallel with him. He does like he is used to – call on the merge tool to integrate the parallel changes – only to get the result: “Do not know how to merge .jpg”. That is when John realises that locking is not always a bad thing.

3. SCM for the company

In the previous chapter, we saw how SCM concepts and principles can be useful and provide help for both individual developers and the collaboration in smaller or larger project teams. In this chapter, we will take a step back and look at the bigger picture – how the project manager, the company and the customer can get service and support from SCM concepts and principles. Even if you are not (or will never be) a project manager or company owner it is still important that you at least know about this picture. As a developer – and as a project team – you always exist in a bigger context. There are other stakeholders and they have other goals and priorities. In order for them to obtain their goals, they may implement processes and procedures that sometimes become part of your daily work and other times something that exist in the background. If you do not know about this and what is the purpose, you will be navigating in the dark and have no idea of how to contribute to the bigger good.

This chapter is based in part on Daniels' book on "Principles of Configuration Management" [Daniels, 85] and in part on the current IEEE standard for "Configuration Management in Systems and Software Engineering" [IEEE Std 828, 2012]. Both address the traditional configuration management activities and see configuration management as an administrative management activity. And for both the word "software" is not very predominant as at the level of abstraction of concepts and principles there is no big difference in whether you are handling software or hardware.

Daniels has this definition of configuration management: "configuration management is a management tool. It is a tool that defines the product, and then controls the changes to that definition. And this is all; the concept of configuration management is not difficult. In a nutshell: you define your product, and you control the changes to that definition or that product" [Daniels, 85]. Daniels also motivates the purpose of configuration management in this way: "If we use configuration management properly, we should know what we *are supposed to build or produce* and we should know at any point in time what we *are building or producing*. And, once we finish, we should know what we *have produced or built*. Configuration management will help us to do that" [Daniels, 85].

The usual order of the four configuration management activities in the standards – and in literature – is: configuration identification, configuration control, configuration status accounting and configuration audit. The reason for this order is probably because they see configuration management as: the base (identifying the product), the focus (change control) and the fringe benefits (status accounting and audit). However, in this lecture note we prefer to group them together in two logically coherent groups: handling information (Configuration Identification and Configuration Status Accounting) and managing changes (Configuration Control and Configuration Audit).

3.1 Handling information

An important part of configuration management is do handle all the data and information regarding a project. Data and information in this context is to be understood in a very broad sense – it covers everything ranging from the physical gearbox of a car to the blueprints for its engine – everything ranging from the source code over the test cases to the technical documentation.

The important "things" (we will be more precise in a moment) regarding the product have to be recorded and preserved – and they have to be made available to whoever might need them.

Configuration identification

The activity called Configuration Identification is where we try to gather all the data and information that is important for a project or a company in order to be able to produce and build its products.

If you have ever been part of a (software) project, you will know that there are many "things" that are created and/or used during a (software) project. Instead of calling them "things" or having to give long lists of examples, we will simply call them *artefacts*. An artefact could be a source code file, it could be an email, it could be the agenda for the weekly project meeting, it could be the code used in an experimental spike to figure out how to make socket communication work or it could be a requirement from the customer.

Some of these artefacts are important for the project, others just serve and are useful for a short while and are only of transient interest. We will call the subset of important artefacts for *configuration items*. Once something becomes a Configuration Item (CI) it comes under the protecting wings of configuration management. We will secure and protect it so nothing bad will ever happen to it – that means no unauthorized, wild, random changes to it, if (approved) changes are made the old version will be preserved – and no version of the CI will never disappear. A CI comes with a cost – configuration management will have to handle it – so we do not want all artefacts to become CIs. So what does become a CI? It depends a lot from project to project and from company to company, but the most obvious CIs are all the parts of the product (and the product itself) – without the engine we cannot build the car, without the source code we cannot build our application. What about the test cases? We could build our application, but we would have no way to be sure of its quality. So all the "supporting" artefacts should also be turned into CIs – and in general all the artefacts where you would be very sad if they went missing. In some cases the binaries from the compiled source code will become CIs, in other cases they remain "transient" artefacts because we can re-build them from the source code in a few minutes and that has a lower cost than paying the configuration manager to preserve it and the developer to find it again.

All the CIs are collected and stored in what effectively becomes a *Configuration Management DataBase* (CMDB). This means that we have one single place where all the CIs can be found – and there is no reason for anyone to make "private copies" of any CI just in case someone might corrupt it – it is in the CMDB and it will remain there forever and it will never change (unless a new version is added). Through the CMDB we can also share CIs between people on the project – so we don't have to email the source code to the testers – and we can share a single copy of the requirements specification with both the testers and the programmers – and if the testers are not happy that the programmers can view (and maybe change) the test cases, we can enforce proper access control to the various CIs in the CMDB. Like a real library of books also the CMDB needs to be *well-structured* so people can find the things they need. In a real library of books, we might not just be interested in the physical book itself, but also information about the book – when was it published, how many pages does it contain, how many copies does the library have, how popular is the book? The same thing goes for the CIs in the CMDB – we would like to know if a version of a source code has been tested or is experimental, we would like version of the requirements specification this test case was created from and so on. So we should have the possibility to store all this information as *meta-data* with the CIs in the CMDB. If we find a well-written book about compilers in the library, we might be interested in seeing if we can connect the author to book that he has written about other topics – or if the book is incomprehensible, we might want to connect to other books on compilers. The same thing goes when we develop a product. We might want to know who is using an interface – or what are the test cases that test this part of the code. The *connections* between CIs should also be stored in the CMDB. So we end up with a CMDB that contains items that have attributes and relations – very much a full-fledged database.

Configuration Status Accounting
Now that we have compiled this wealth of information we have to put it to good use and make relevant information available to all the people in the project – that is what happens in the activity Configuration Status Accounting.

In the good old days the primary stakeholder for using the information was the project manager and he was given relevant information in a bi-weekly report. Modern Configuration Status Accounting seeks to cater for everyone in the project – programmers, testers, marketing, QA and so on. Furthermore, information that is given in a bi-weekly report very soon gets out of sync with the actual situation. Therefore the modern way focus more on real-time data drawn directly from the CMDB in the moment that it is asked for. So much of what goes on in this activity is about formatting and putting together single pieces of data into information that makes sense and is useful for different groups of people in a project. In some cases this has the form of “pre-defined queries” on the CMDB in other cases the CMDB is made accessible for self-service through ad-hoc queries.

3.2 Managing changes

In the previous section we defined the product and all the supporting things needed to develop the product – the Configuration Items. It was also hinted that a CI might exist in more versions. There are several reasons for this – we often don’t get it right the first time, we need/want to port the application to a new context – or we get new ideas to functionality that can be added. In any case there will be a lot of changes to the initial definition of the product and the activities of Configuration Control and Configuration Audit make sure that this does not end up in chaos and anarchy and total confusion.

We need to have a well-defined process for how changes are created and how they will be handled – and we need to have a way to make sure that the results of changes are not accepted if they are not of the agreed upon quality.

Configuration control
It is in the activity Configuration Control that we define how we want to handle changes.

Any wish for a change should start with a definition of the change that is wanted – we will call this a *Change Request* (CR). Requests for changes can have many different sources – it could be a tester signalling a problem – it could be a user informing about a bug – it could be a programmer telling about an ambiguity in the requirements specification – it could be marketing hyping the latest functionality from our competitor and that we should also have. In some cases that will give rise to different CR forms because different kinds of information is needed to process properly the CR, but in simple contexts one unified CR form will often be sufficient. The CR will go through a number of processing steps before it ends up on a *Change Control Board* (CCB) meeting where it is decided whether we should do the change or not. Let us look a little close at the *processing* that has to be done before the CCB. First we need to do some sort of *filtering* on the CRs. We need to make sure that all the needed information is there on the CR form – both information needed to take a decision at the CCB meeting, but also information need for the programmer and tester to implement the CR. In many cases the originator of the CR will not be easily accessible to get that information later on. Another filtering that has to be done is the removal of duplets – that is CRs that we have seen before. Most often that will be bug-reports from different users, but still concerning the same bug. We do not want to treat them as single CRs, but we might still want to capture information about how many times a bug has been reported on the CR form. After the filtering we have to carry out an *impact analysis* on each CR to estimate the consequences of accepting it. If we change a requirement it will have consequences for the source code and the test cases. If we fix a bug, it may have consequences for the places in the code that had a work-around for the problem. If we implement a new feature, it may have consequences for the architecture or the efficiency of the application. Basically we need to know what are the costs of accepting a CR. Now we have the information necessary to proceed to the CCB meeting.

The purpose of a Change Control Board (CCB) is to make *decisions* about which CRs are *accepted* for implementation, which CRs should be *rejected* – and which CRs we do not have sufficient information about to be able to make a decision about now and are therefore *deferred*. If we had infinite time and resources we would not need a CCB and would have the luxury that we could accept all CRs. However, that is never the case in real life – so we have to carefully prioritize where time and resources are used. There are good reasons why it is called a change request and not a change order. Some bugs might not be that critical (or only to a few customers), some features may only be scratching an itch without really creating revenue. The CCB meeting is where the *chairman* will look at each single CR and based on the information that has been compiled about the CR will make a dictatorial decision whether to accept or reject the CR. Often the chairman is the project manager, but in some cases it is a customer, a product owner or sales acting as a proxy for the client base. The chairman acts as the captain on a ship – a wise captain listens to input from his crew, but he alone makes the decision. Usually a project manager is the person with the largest base of information about the project and is therefore the best suited to make a decision. The information that he may lack can be provided either through the impact analysis (if it is simple) or an expert can be invited to the CCB meeting and the chairman can discuss the CR with him to be better informed. It is not unusual to process hundreds of CRs on a two-hour CCB meeting, so not all CRs need a lot of discussion and thinking – if a tester has flagged a bug as critical and we have plenty of time and resources before the next release such a CR will be accepted in two seconds.

After the CCB meeting we need to *further process* the CRs. If we want to be nice, we will inform the originators of rejected CRs of the outcome – sometimes with a motivation, but in any case they will know what happened to their CR. For the accepted CRs we need to initiate and follow the implementation of them. We need to schedule people for the different steps in the implementation and we need to make sure that these steps have been carried out – that the tester wrote test cases, the programmer wrote code, the tester tested it and so on. The final step when everything has been done and the implementation has been accepted is that the CR should be flagged as "closed". The project manager – being responsible for managing the resources on the project – might be interested in following the global state of the CRs – how many are "open", how many are "under testing", how many did we "close" last week" – such figures and answers will be provided as part of the Configuration Status Accounting described above.

Configuration Audit

We are approaching the release date and want to be absolutely sure that we deliver a well-functioning and complete product – and that is when we will do a formal Configuration Audit.

The purpose of a Configuration Audit is to verify that all the physical parts of the product is there. So if our product is a car, the parts could the wheels and the owner's manual. This is called a Physical Configuration Audit (PCA). We also want to verify that all the functionality that we promised to deliver is actually there. So if our product is an application the functionality could be the bug-fixes and features we promised – I short, the accepted CRs. This is called a Functional Configuration Audit (FCA). From the approved Configuration Audit we will establish the new *baseline* for the product and its supporting CIs. This baseline will remain fixed until the next Configuration Audit and will provide us with something stable to work against, so it will not feel like shooting at a moving target.

The result from the Configuration Audit will be a great help to the project manager to know if the project is ready to release – and the project manager is also the person who will act on that information. Even if the outcome of the Configuration Audit is not perfect, the project manager might decide to release anyway. It could be that a bug-fix has been implemented but not tested and quality assured – but if the project manager knows that the bug isn't critical and that the programmer fixing it usually does a good job, he might decide to proceed with the release anyway because the company has already booked and paid for a lot of commercials for this new release.

The concept of a *baseline* can also be put to good use in a more informal way. In practice anything – not just the product – can (and should) be baselined. The requirements specification, the source code, an interface, the technical documentation and so on. We should baseline anything that we would like to remain stable for some time and only change when we have assured that the change is of a certain quality. Again the purpose is to provide a stable ground on which to work. Just like for the baseline for the product we should have a process for managing changes – Change Requests and a Change Control Board. However, since these baselines are more informal also the process for managing change will be more informal and new baselines will probably be established more frequently than for the product. These baselines are well-defined reference points – snapshots at a given time – that will never change, but maybe get a new version. While you are working on creating the new baseline – of e. g. the source code – your code is work in progress and obviously you can change a typo in a comment or rename variables without having to file a change request. However, once your change has been promoted to a baseline and frozen even changing a typo in a comment will require a change request since it changes the stable baseline that other people will rely on.

4. References

[Babich, 86]: Wayne A. Babich: *Software Configuration Management – Coordination for Team Productivity*, Addison-Wesley Publishing Company, 1986.

[Bendix&Vinter, 01]: *Configuration Management from a Developer's Perspective*, proceedings of the EuroSTAR 2001 Conference, Stockholm, Sweden, November 19-23, 2001.

[Daniels, 85]: M. A. Daniels: *Principles of Configuration Management*, Advanced Applications Consultants, 1985.

[Feiler, 91]: Peter H. Feiler: *Configuration Management Models in Commercial Environments*, Technical Report, CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie-Mellon University, March 1991.

[Garousi et al., 19]: Vahid Garousi, Görkem Giray, Eray Tüzün, Cagatay Catal, Michael Felderer: *Closing the gap between software engineering education and industrial needs*, IEEE Software, 2019.

[IEEE Std 828, 12]: IEEE Computer Society: *IEEE Standard for Configuration Management in Systems and Software Engineering*, 2012.

5. Further reading

[Sommerville, 11]: Ian Sommerville: *Software Engineering – Chapter 25: Configuration Management*, ninth edition, Addison-Wesley, 2011.

[Milligan, 03]: Tom Milligan: *Better Software Configuration Management Means Better Business: The Seven Keys to Improving Business Value*, Rational/IBM white paper, August 2003.