

SCM in a Large-Scale Agile Development Project

Paul Dyson and James Spalding

paul.dyson@e2x.co.uk

james.spalding@e2x.co.uk

Introduction

This paper presents an experience report of a large-scale agile development project, the specific software configuration management challenges faced, and the resolution (or otherwise) of those challenges. The authors were the technical lead and project manager for the project which consisted, at times, of over 60 technical staff: architects, developers, QA engineers, database administrators and so on.

The project ran for a total of four years, with a number of identifiable sub-projects taking place over that time, and the source control system remained largely unchanged for the whole of this period. However, a large number of bespoke processes and utilities had to be developed over the lifetime of the project to deal with SCM-related issues.

The Project

The objective of the project was to build a 'global e-commerce platform' for a major multi-national corporation. The purpose of the platform was to provide a foundation for all regional websites for both business-to-consumer and business-to-business information and servers. As such it had to integrate a number of supply-chain, product management systems and logistics systems; had to allow for significant regional customisation; had to provide mechanisms for marketing campaigns to be introduced with little or no technical involvement; and had to support all of these in multiple languages and currencies. The platform was to be supported by an extensive content management system, administered by a global team but used by individual regional organisations.

Despite the size of the system and the development team (just over 40 technical staff at the outset), an agile development process was selected. This was for a number of reasons:

- Although the high-level requirements were clear, detailed requirements were incomplete and, in many cases, were the subject of negotiation both

within the customer organisation and with their suppliers. In effect, the requirements set was constantly changing.

- Even where there was agreement on the requirements, priorities often changed due to external events. What was considered to be a low priority requirement could, without warning, suddenly become very important.
- Even with a fairly large team, this was clearly a long-term project. However there was an urgent need to replace the current inflexible and limited web sites with more powerful. The customer organisation was concerned they were losing ground on the web to their competitors.

An agile development process offered the following solution to these problems:

- Get started quickly, with the minimum of upfront architecture and design, and release a small, but production-ready, version of the system in a short space of time. Spend some of the upfront time saved in concentrating on creating automated tests that would enable refactoring to 'better' designs, as what 'better' meant became clear.
- After the initial release, evolve the production system over a number of short iterations. At the end of each iteration, take the new system version live.
- Only prioritise requirements shortly before the start of each iteration. Only detail the high-priority requirements as they are about to be implemented, ensuring that exactly what was required at the time of implementation is delivered. Review nightly system builds to ensure that requirements reflect what was actually wanted.

Although these kinds of benefits have been realised on much smaller scale projects, an agile approach to building much larger systems with larger teams was still unproven. In implementing the agile development process, a number of practices that work well with teams of around eight developers had to be extensively modified to scale up to work with 60+ technical staff.

The Process

The project operated a 'two-tier' iterative process. The 'external' iteration was ten weeks long: the business prioritised about ten weeks worth of requirements, which were then planned in detail by the development team. The business and development team then agreed which requirements were to be delivered (the business had the chance to change priority on the basis of the estimates given to them by the developers). At the end of each iteration the development team delivered a version of the system that could be taken live if the business

approved it. Not every version of the system was taken live due to changing business pressures and external factors (such as translation of content or the signing of contracts) but every version of the system was fit to be taken live if the business so wished.

'Internally' the development team worked to one-week iterations. Each iteration started with a short verification of the requirements and estimates and initial design workshops where necessary. Planning and design could usually be completed in an afternoon. The development team practiced continuous integration with every developer delivering their code back into the development codebase at least once every day, and often much more frequently than that. There were at least five automated full builds every day and the running of both automated and manual tests was a continuous activity.

At the end of each week-long iteration, a 'release candidate' version of the system was produced. The idea of a 'release candidate' was that it should be technical capable of being taken live, even if it didn't necessarily make good sense to do so. For example, a release candidate produced early in the ten-week iteration might contain a basic product search function but none of the advanced search options required. The search in the release candidate must be fully working but did not have to implement all the search requirements expected to be delivered at the end of the ten-week iteration.

The production of release candidates served two purposes. Firstly, it allowed the business to alter the 'external' iteration length if required. Although this was actively discouraged, there were a few occasions where shortening an iteration to deliver some functionality early, or lengthening an iteration to enhance some of the functionality a little before go-live, was deemed by necessary by the business. In the example above, the business could decide they wanted the release candidate with the basic search implementation to go live because the system version also contained some newly-important functionality which was far more important than advanced search capabilities. The second purpose of the weekly release candidates was to focus the development team on continuous delivery of high-quality software, reducing the risk that the end of a ten-week iteration would result in a scramble to fix bugs and integrate work.

After the go-live of the first business-approved version, maintenance of the live system and development of the next version of the system were carried out in parallel. At the end of a ten-week development iteration, the last release candidate went into formal QA. Because QA was a continuous activity, this period was a genuine formality: signing off bugs that had been fixed during the development iteration and re-prioritising outstanding bugs (either raising the priority so they were fixed during the QA process or lowering the priority so they would be fixed in the next iteration). Reprioritisation of the bugs was approved by

the business: no bug-fix could be deferred to the next iteration without the business's approval. QA was always scheduled to take two weeks and this was only lengthened by a day or so on a couple of occasions.

At the end of QA, the final release candidate was deployed into the production environment. Any defects identified in the production system were prioritised by the business as 'urgent' or 'non-urgent'. Urgent bugs were fixed by a small dedicated maintenance team and applied to both the production version of the system (with subsequent QA sign-off and go-live) and the version in development. Non-urgent bugs were fixed in the development version only by either the maintenance team or the development team depending on the nature of the bug and the current development work being carried out. At the point where a new release candidate went into formal QA, a moratorium was called on defect fixing in production: all defects were fixed only in the new release candidate.

SCM Problems and Solutions

Tool Choice

The first problem we faced was that of tool choice. Given the emphasis on an agile development process, the tool primarily needed to support team agility. Our high-priority requirements were:

- **Speed:** Check in and checkout had to be fast. Tagging could be much slower (only required for automated builds).
- **Scalability:** The tool needed to be as fast for 60+ developers as it was for just a few.
- **Simplicity:** The tool needed to be simple to set up, administer and use. We had no resource for full-time SCAs to support both the tool and users of the tool.
- **Flexibility:** The tool needed to adapt to our changing process.

The tool we settled on was CVS. This is clearly a surprising choice from an SCM point of view, especially as no-one would claim that CVS is an SCM tool; it provides pretty basic version control. However, CVS was the de facto choice for agile projects at the time (although Subversion has become more common) simply because it is pretty fast if you do all your work on the trunk (which is the case if you practice continuous integration), it scales reasonably well, it is extremely simple to set up and administer and, being so basic, it is extremely flexible (it does version control and the rest is up to you).

Although other commercial and non-commercial tools offered similar performance and scalability (within our requirements), no other tools could match the simplicity of CVS. One key requirement of an agile project is that the SCM tool is 'almost invisible' to the developers. There should be no complexity around starting work on a task or delivering that task. Ideally check-in and check-out should be 'one button' operations with very simple merging operations. One developer's work should not impact on any other developer's work (no reserving of files). In these areas, CVS had the advantage over the other tools considered, particular with its tight integration with the standard development environment for the team (Eclipse). Ease of use by developers was match by the simplicity of setting up the server on extremely modest hardware and ease of administration.

The other critical factor in settling on CVS was flexibility. We knew that we would encounter problems with SCM-type operations (as described below) but any tools that provided complete or partial solutions to those problems did so by introducing additional tools and requiring particular ways of working that didn't fit well with our agile process. From our point of view SCM practices should fit into our development process, not the other way around.

Maintenance

Although all the main development work was carried out on the CVS trunk, a branch of the trunk was made each time a release candidate was taken into the formal QA process described above. At this point any defects identified during QA needed to be fixed on the QA branch and the new development trunk. On two occasions, however, a defect was identified in production during the QA process that was deemed to be urgent and so had to be fixed on the production branch, the QA branch and the development trunk.

At the end of the formal QA period, the QA branch was taken live, becoming the new production branch, and the old production branch was retired. From this point forward, urgent defects were fixed on both the production branch and the development trunk, with non-urgent defects being fixed in the development trunk only.

The problem we faced was how to identify the changes made to fix a particular defect. The typical approach to making changes that have to be merged into more than one system version is to take a 'task branch' from one version, make all the changes required to fix the defect in that task branch, and then merge the task branch into each of the branches (in our case production branch and development trunk) where the fix is required. However, this approach does not fit with continuous integration: a defect fix might take a number of days and so there will be several 'integration points' during the fix, and several different developers or pairs of developers will work on the fix.

Several tools offer extensions to the task-per-branch approach to support multiple integration points and to cope with having multiple developers work on a particular task but all require adoption of additional tools and proscribed processes that we felt were too heavyweight to fit into our agile process.

In order to solve this problem we had to adopt a convention for check-in comments so we could identify changes made for a particular defect such that defect fixes made within the production branch could be applied to the development trunk. This proved to be sufficient but we had develop our own scripts to identify all the changes made to the production branch for a particular defect so we could identify what to merge into the production trunk.

‘Selective Assembly’

One of the characteristics of a codebase that has been developed using continuous integration is that it is ‘monolithic’. That is not to say that there is a single ‘blob’ of code with no discernable architecture, but that the advancement of the codebase is governed by time (all of the code developed today is merged in) rather than by function (all of the code that implements requirement 1234 is merged in). For the vast majority of the time, this was fine, but there were occasions where, towards the end of an external iteration, things had changed and the business wanted either wanted to remove some completed functionality, or wanted to de-prioritise some incomplete functionality in favour of spending more time on some newly-important requirements.

The problem, with a monolithic code base, was how to identify all the code for the function to be removed or ‘capped off’ (our term for ensuring that incomplete functionality was inaccessible by the user). Our ideal solution to this problem would have been to be able to map all requirements to the code that implemented it and to create system builds by assembling only the code that implemented the requirements to go live; leaving out the code for the partially-implemented or removed requirements.

We did review one tool that provided this facility but rejected it on the grounds that it mandated that all requirements be managed in a particular (heavyweight) tool and that a particular development style was adopted (the tool wasn’t able to cope well with refactoring for example).

In the end we didn’t find a solution to selectively assembling the code and had to resort to a manual process for identifying code related to a particular requirement so it could be removed or ‘capped off’. Interestingly, our test driven approach didn’t provide a great deal of support: it is hard to prove that a test is failing because all the code that should makes it pass has been removed rather than partially removed.

Artefact Dependencies

The days when a 'system' consists of source code in a particular programming language and a database have long gone. Our system consisted of web page templates, web service definitions, Java services, object-relational mapping schemas, database schemas, application server configuration information, binary assets (images and documents), messaging definitions, etc., etc. Some of these were produced and managed by the development team, many of them weren't.

There are many and varied dependencies between these artefacts. For example, a page template produced by the development team may reference an image produced in the content management system by an external graphic designer. This reference may be direct (embedded in the template) or indirect (determined from a database table via a service that provides internationalisation). As another example, a new Java service may require some new data to be persisted, which also requires that the object-relational mapping layer, the database schema, and the static dataset be updated.

When we were creating a system release, the system had to consist of the correct version of all these artefacts. But how do you determine whether you have all the correct versions of none source-code assets (for which to can perform a full build to ensure all the code compiles)? You can run your entire test set but you have to write a lot of tests if you want to ensure things like references images are present. It may also not be suitable for all artefacts to be managed within the same repository: CVS wasn't much of an SCM tool but it certainly wasn't up to being used as a content management system. So the dependencies between artefacts cannot all be identified by doing build against your SCM repository.

Is this strictly an SCM problem? We don't know – but our project suffered several times from not being able to identify all the artefact versions we required to deploy a new system release. Traditionally, building system releases has been the remit of SCM tools but no tool we have seen can cope with the intricacies of this wider definition of 'system' and we had to carry out a lot of testing and manual verification to ensure all artefact versions were in place. We believe this is a problem common to all projects, perhaps emphasised by the agile nature of our project which resulted in a greater-than average number of system releases over the four years we were involved with the project.

The 'Agile SCM' Problem

Although existing tools have been upgraded, and new tools have appeared, in the four years since we chose CVS as our 'SCM' system, we don't believe that there has been much progress in agile SCM. If we were running the project

again, we might choose SubVersion as a 'better CVS' but we'd still be unlikely to choose a 'proper' SCM tool. Our view is that SCM is a practice that should be integrated into a development process rather than one that defines what development process you can use. The SCM problems faced on an agile project are really no different than those faced on a project using a more 'heavyweight' process; but that the constraints on what you can do to solve those problems are different.

Fundamentally, we believe this is a tools problem. The constraints on solving problems like identifying which artefacts have changed to fix a defect and selective assembly are imposed by the tools, not by SCM practices. There is a need for SCM tools that acknowledge and work with agile practices such as continuous integration and refactoring in a simple and flexible manner.

About The Authors

Paul Dyson and James Spalding are both consultants for e2x limited – a consultancy specialising in agile development processes. Paul Dyson was one of agile development's 'early adopters'; running the first eXtreme Programming (XP) project in the UK in 1998. Paul is the co-author of "Architecting Enterprise Solutions: Patterns for High-Capability Internet Systems" published by Wiley & Sons in 2004.

James Spalding has over 20 years experience working as a Project manager in the IT industry, both for small 'dotcoms' and large corporations. Over the last 5 years he has focused on managing internet technology projects using agile development methodologies. Varying in size from 5 to 50 people, these projects covered the complete software lifecycle from requirements capture, through project delivery to support and maintenance.