CM Crossroads
the Configuration Management Community

The Illusion of Control by Steve Berczuk

by **Steve Berczuk, Robert Cowham, Brad Appleton**

Effective environments arise out of a group of people using approaches in the correct context. What seems to be a good solution in one environment will cause pain in another because some of the elements that made the solution a success in one situation were missing in the second. This is what patterns and pattern languages are all about. This month we explore why it is important to understand the context of Agile techniques when you are trying to build a more agile SCM environment, and how people can fool themselves into thinking that their non-agile environment has more control.

Control

Figure 1: Control room, waterworks. Conduit Road, Washington, D.C.

From time to time I give talks to groups of software developers at various companies about Software Configuration Management Patterns and Agile SCM. Two of the most common frustrations that people express about their development process are:

- Instability in the codeline: The latest code does not compile, or pass tests.

- Very long pre-check-in processes. Either there is a long series of hurdles to check in code such as tests or reviews, or the team uses an Integration Token in a way that means that there is a long wait to do any check-in (or both).

What is amusing is that you often hear about both of these issues from people in the same organization at the same time! The integration token or extensive test suite was started as a way to address the code instability problem, yet the end result is that neither output nor quality improves in any real way.

The heart of the problem here is a classic software development problem: that Integration is difficult. Slowing down the check-in process gives the illusion of control, but that is only a short term fix.

What often happens is that after a being frustrated by a long line of broken builds due to testing and check-in controls that are too loose, the team swings the process the other way towards heavy process and control applying the theory that slower is safer. The problem is that the slower process still permits breaks from time to time, while encouraging larger integration intervals. These larger integration intervals break the team’s rhythm and also increase the risk that there will be problems come integration time. (For more about the role of Rhythm, see [1])

A Story
Here is a common situation in many organizations. After a series of dark and stormy release cycles during which the release is late, and has poor quality, someone in the team decides that the way to fix the chaos is to impose more control on the development process. If the problem is poor quality, the answer is to do more testing. (The proposed answer is rarely “to test more effectively” except in the sense that “more” is often viewed as “better.”) The rule comes down that before check-ing code in, developers need to run a long (or very long) suite of tests.

The quality of the codeline does improve for a while, and developers are happy that they are spending less time working with broken code. Velocity increases, and developers start adding features. There is more change in the codeline, and this change is good. Then strange things start happening. Otto makes some changes, runs the test suite, which by now takes 90 minutes to run. Actually he makes a lot of changes since it takes a while to run the tests, and he wants to batch as many changes into his check in as possible. Unknown to Otto, Sue has also started making some changes. Both Otto’s and Sue’s test pass and they check in their code. Brad updates his workspace from the codeline and discovers that the code does not compile, much less pass tests. What has happened is that Otto and Sue made incompatible changes. Because the tests took so long to run, they made lots of changes, increasing the chances that the changes would conflict. Figure 2 illustrates this.

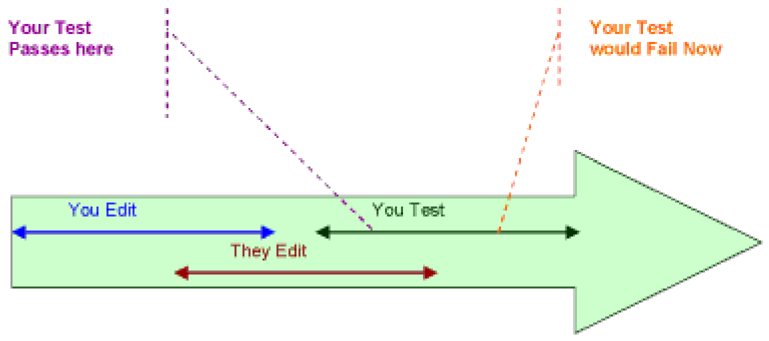

Figure 2: Phase Shifts and Testing

The team's manager, Beth, sees a problem in need of a solution and declares that henceforth the team shall use an integration token. Only one person shall perform a check-in at a time. The check-in process is:

- Grab the integration token.
- Update your workspace from the codeline, resolving any conflicts.
- Run the test suite.
- When the test suite passes, check in the code.
- Pass the integration token on to the next person in the queue.

The problem of conflict is resolved, but now Beth notices that the rate at which features are being added to the code is slowing down. Only one person can do a check-in every 2 hours or so, which means that, at most, 4 people can check in changes at once now. In reality it may be less because there will always be people with commitments after work meaning that they should not even start the check-in process less than 2 hours before they need to leave.

While trying to fix the immediate problem, Beth created a longer term problem. A lack of quality leads one to want more control, which reduces quality in another dimension. Figure 3 illustrates this. For some applications with critical systems and small teams, this approach may work well, but in many cases the dominant business need is to change the codeline to add features.

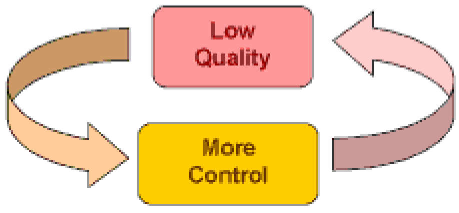

Figure 3: Feedback between control and quality

How do we fix this? We can follow both lean and agile principles (which, at their heart, are similar) and try to address the core problems: Facilitating flow and eliminating waste. Some of the SCM Patterns address this.

Continuous Improvement in Context
Beth took a situation that was out of control and tried to control it, applying the theory that slower is safer. In some cases that's true. Slower often gives you more chances to correct yourself and react, but slow might not be safer in all cases. When driving a vehicle, it often appears safer to drive slow. Yet sometimes driving conditions such as mud or sand create a "viscous cycle" where it becomes vitally

important to maintain a minimum threshold speed, otherwise you get stuck in a rut and can’t get out of it.

As we said last month ([2])

> Heavyweight configuration management environments that make developers wait non-negligible periods of time for things like check-out-authorization, presenting and approving all files to be modified prior to coding, protracted build-and-test cycles, waiting for a new baseline to be promoted for use, etc. are considered detrimental to the health of an agile development environment.

To solve the real problem consider what your goals are. If you want an Active Development Line, you want a codeline that is in good shape most of the time, but rapidly evolving. A codeline that never changes will always be safe, but you are not adding much business value.

The SCM Pattern Language shows you one way to resolve these issues. Figure 4 shows the relevant patterns. You create an *Active Development Line* by having developers work in a *Private Workspac*e. Before anyone checks code in to the codeline, developers must do a *Private System Build* and run a *Smoke Test*. The *Smoke Test* should quick to run, even if it is not comprehensive.

At this point you are probably asking yourself how it is possible that a less than comprehensive test can give you any sort of stability. This only works because developers will be running more comprehensive *Unit Tests* while they are doing coding, and you also have an *Integration Build* process that runs a full suite of *Unit Tests* and a *Regression Test*. You are making the decision that delayed integration will cause more problems than a slight chance of a build problem. And the integration build and test process ensures that when a problem slips through, you will catch it quickly.

In many cases, the full suite of Unit Tests may be quick enough to run so that developers can, in fact, run comprehensive tests before check-in, but if the comprehensive testing starts to take long enough that developers defer integration, you should consider a simpler pre-check-in test suite.

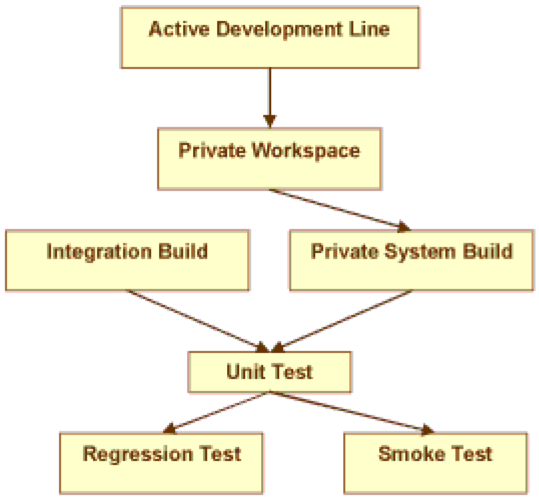

Figure 4: Supporting an Active Development Line

Agile, Lean, and other Cliché's
The SCM Patterns support a development environment that adapts to change. Most all development environments are ones where change and uncertainty is the rule. There are two adjectives that describe approaches to development that are focused on change: Agile and Lean. Let's spend a few minutes reviewing the concepts. If you want more detail, have a look at the November column ([2]).

Common Objections
Here are some common objections we encounter, and some answers.

- **We can't integrate frequently**; some of our software is intellectually difficult, and it may well be weeks before code is ready. While the core code may be difficult, it is probably possible to define the interfaces to components so that related systems can move forward, as in Figure 5. If the developer of the really complex system is writing appropriate unit tests that pass, the integration overhead can be minimal.

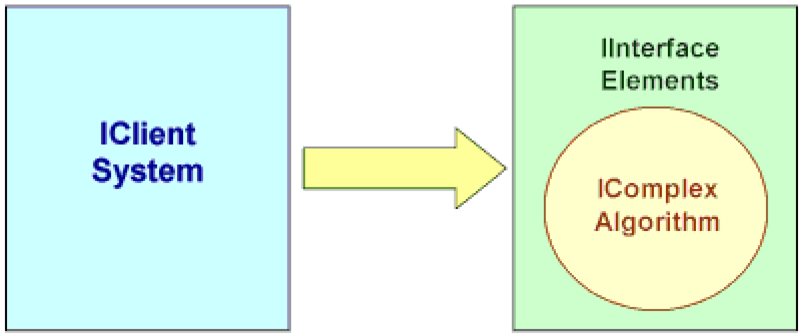

Figure 5: Testing with Complex Algorithms

- **I can't write tests unit tests**. Are you sure? In many cases, "not testable" often indicates a potential problem with the architecture.

- **I don't need to be agile**. If you are delivering software that gives value to your customer in a reasonably timely fashion, and with reasonable quality, then you are probably doing everything OK. You may find benefit in working to improve your processes so that things are even better. If you are really are doing an effective job, it is extremely likely that you are open to new techniques.

- **We need to define requirements in advance, so this iterative/incremental approach isn't going to work for us**. Ask yourself when the last time you worked on a project where everything you planned up front was accurate. Agile approaches don't say "Don't Plan." Agile approaches simply acknowledge that plans are often incomplete or incorrect.

- **Our Developers are doing agile, but Release Engineering needs control**. There are two points to here: 1. See the title of this article. 2. You should think of your self as working in a Software Development Organization, not a Release Engineering group, or a Programming group. If one groups needs don't mesh with the others you'll have integration issues that will slow things down. Try to understand how to work as a team.

Continuous Integration

Rather than control, what you need is visibility. Continuous Integration (or at least frequent integration) is an excellent technique for giving you a view into the state of the system For more about Continuous Integration across multiple component teams, see our article from March [3]

Conclusion

When a process is broken we are often tempted to do *something*, even if we're not sure that it is the right thing. By imposing rules and processes we feel like we have control, but more often than not, the control is illusory. And in many cases the solution is very obvious; to solve integration issues you need to integrate code frequently. Most errors occur at boundaries, so many problems are integration problems at the code.

Sometimes doing this will clash with established process or culture. You need to decide if you want to change things and if you do want to change things, you need to do things. To quote Yoda: Do or Do Not, there is no Try.

Pointers and References

We discuss integration tokens in this article. For a discussion of when an Integration Token makes sense, read our November 2003 article, *Codeline Merging and Locking: Continuous Updates and Two-Phased Commits* (http://www.cmcrossroads.com/cgi-bin/links/jump.cgi?ID=1374).

The photo in Figure 1 is by Edwin Rosskam, Library of Congress, Prints & Photographs Division, FSA-OWI Collection, [LC-USF34-015836-E DLC (b&w film nitrate neg.)]

1. Dikel, D.M., D. Kane, and J.R. Wilson, *Software Architecture: Organizational Principles and Patterns*. 2001, Upper Saddle River, NJ: Prentice Hall.

2. Appleton, B., S. Berczuk, and R. Cowham, *Tasks and Branching Patterns*, in *Configuration Management Journal*. 2004. http://www.cmcrossroads.com/article/35521

3. Berczuk, S., B. Appleton, and S. Konieczka, *Continuous Staging: Scaling Continuous Integration to Multiple Component Teams*, in *CM Crossroads Journal*. 2004. http://www.cmcrossroads.com/newsletter/articles/agilemar04.pdf

---

Steve Berczuk is an Independent consultant who has been developing object-oriented software applications since 1989, often as part of geographically distributed teams. In addition to developing software he helps teams use Software Configuration Management effectively in their development process. Steve is co-author of the book ***Software Configuration Management Patterns: Effective Teamwork, Practical Integration***. He has an M.S. in Operations Research from Stanford University and an S.B. in Electrical Engineering from MIT. You can contact him at steve@berczuk.com.

Brad Appleton is co-author of ***Software Configuration Management Patterns: Effective Teamwork, Practical Integration***". He has been a software developer since 1987 and has extensive experience using, developing, and supporting SCM environments for teams of all shapes and sizes. In addition to

SCM, Brad is well versed in agile development, and cofounded the Chicago Agile Development and Chicago Patterns Groups. He holds an M.S. in Software Engineering and a B.S. in Computer Science and Mathematics. You can reach Brad by email at brad@bradapp.net

Robert Cowham is the founder of Vaccaperna Systems providing SCM consultancy and training to organisations. With 20 years of experience in software development, he has long had an interest in SCM, and has worked with clients around the world in this arena during the last 7 years. He is on the committee of the CM Specialist Group of the British Computer Society for whom he has organised several events, including their 2-day conference in 2003 (and their upcoming 2005 event). He has a BSc in Computer Science from Edinburgh University and is a Chartered Engineer (CEng MBCS CITP). You can contact him at rc@vaccaperna.co.uk