Editors: Andy Hunt and Dave Thomas ■ The Pragmatic Programmers
andy@pragmaticprogrammer.com ■ dave@pragmaticprogrammer.com

Pragmatic Software Configuration Management

Steve Berczuk

Whenever I start a new software project, I look at the infrastructure the team has in place for doing things like setting up a development workspace, building, doing version management, and so on—the things that belong to the larger topic known as software configuration management (SCM). I worry about these elements because, in a sense, I'm lazy. I want to direct my energies into producing code that works and is maintainable. When I have to spend a lot of manual effort dealing with infrastructure issues, I suspect that I'm not generating as much value for the organization as I would if I could focus more on the steps for writing code (this includes understanding requirements and testing). Workspace management, integration, and build are important developer tasks, but they shouldn't require excess effort.

Often an organization's SCM mechanisms don't help with the work of building software as much as they could. Either no mechanisms exist for doing common tasks or the processes to use those mechanisms are too complicated and become tasks in themselves. On a daily basis, developers shouldn't notice SCM that much, and what they do notice, they should eagerly embrace because these things help them do their jobs. When developers don't find SCM processes to be helpful, it's often because the processes don't serve the organization's goals well. This tends to be due to two issues: first, basic SCM structures are lacking and, second, the structure doesn't fit well into the development environment. In our book *Software Configuration Management Patterns: Effective Teamwork* (Addison-Wesley, 2003), Brad Appleton and I saw this as a perfect application for pattern languages (see the related sidebar). Here is a brief overview of the essential patterns for a basic, agile SCM environment.

Codelines and policies

The first thing software developers should do before providing a solution is to be sure that they understand the problem (see D.C. Gause and G.M. Weinberg's book *Are Your Lights On? How to Figure Out What the Problem Really Is*, Dorset House, 1990). Two items—codelines and branching, and stability—are especially relevant for this task.

A *codeline* is a progression of the set of source files and other artifacts that make up some software component as it changes over time. A codeline has a purpose associated with it, described in the *Codeline Policy*. A codeline that forks off from another codeline for the purposes of parallel development is often called a *branch*.

When confronted with a version control tool, you might be tempted to branch frequently to isolate development work efforts. Branching codelines that might need reconnection later will require merging, which often works well with proper tool support but can be error-prone. Merging is a reason many people swear off branching—even when it makes sense—or shun version control tools. In many cases, most development can occur on a single development line, a *Mainline*.

An SCM Pattern Language

A *pattern* is loosely defined as a solution to a problem in a context. Patterns are proven solutions. *Pattern languages* are a grammar that describes how to fit the patterns together to make something meaningful.

The software configuration management pattern language describes how to build a development environment that will have the correct amount of SCM infrastructure to create software more quickly and reliably. Of course, SCM is only part of the picture (if an often neglected part); you still need good development and coding practices. But the SCM pattern language will help you solve problems so you can focus your energy on more difficult tasks.

Figure A shows a map of the portion of the pattern language I discuss in this column. Each box represents a pattern. Each pattern's name tells you the structure you should create (Mainline, Release Line, Private Workspace). You'll need to read the pattern to know more details about how and when to apply it, but the pattern names form a useful vocabulary for discussion. In our book, *Software Configuration Management Patterns: Effective Teamwork,* Brad Appleton and I have grouped the pattern language into two sections: patterns about workspaces and patterns about organizing codelines. In the interests of space, the main text provides an overview of the workspace patterns after a brief description of some key codeline patterns.

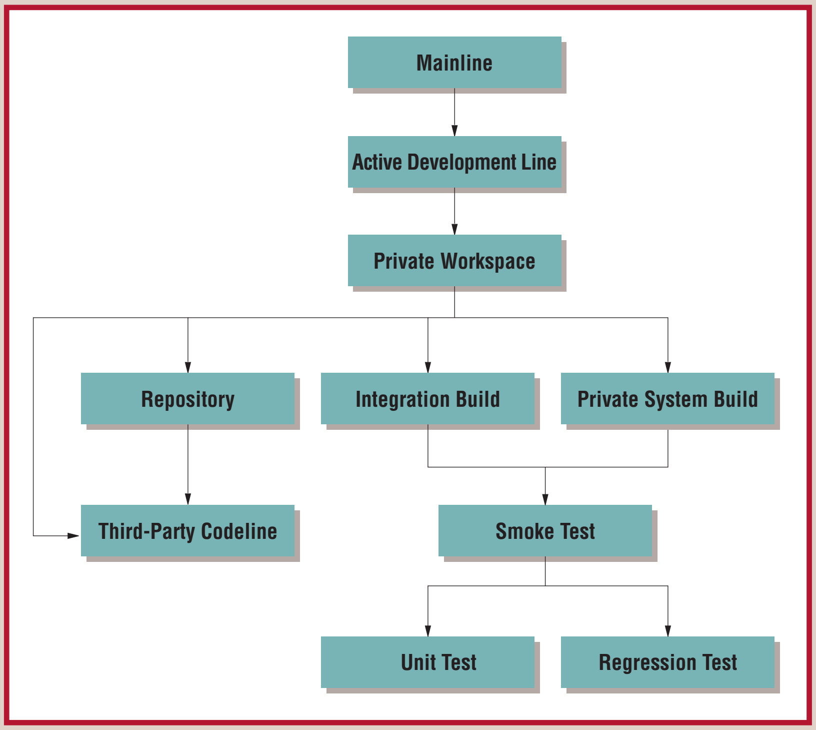

Figure A. Part of the software configuration management pattern language. An arrow pointing from pattern A to pattern B means two things: you apply pattern B when pattern A already exists and pattern B helps pattern A work.

We often think about SCM processes as controlling change tightly and ensuring stability. In many situations, this is not necessary. Each development stream (or codeline) should have an appropriate Codeline Policy that describes what pre-check-in tests to run and so on. Because many applications don't need 100 percent ready-to-ship-at-any-time stability, we can establish a Codeline Policy that gives us an *Active Development Line*, where we slightly favor progress over stability.

Of course, at some point we'll want to have code that you can ship at a moment's notice. Once we ship a version of a product, for example, we might want to allow for changes to make patch releases. This code should be on a *Release Line*, which is a codeline with a Codeline Policy that favors stability over progress.

Getting started with development

Managing change is a universal problem in software development. On the whole, change is good; without it, nothing gets done. However, developers need some control over their development environment. A *development workspace* is where developers put artifacts that they need to code, build, and test. These artifacts—such as source files, libraries, and configuration files—can be shared (in whole or in part) or private to each developer. In a shared workspace, any change instantly affects other developers using the workspace. Normally, it's good to be in synch with your team, but most people need a window of time to work free of change. Giving developers each a *Private Workspace,* where they can integrate changes appropriately and build the whole system if needed, provides this.

You can use a *Repository* to populate a Private Workspace with the right versions of the things that a developer needs. The Repository is the central place where all project components reside, providing "one-stop shopping" for setting up or updating a workspace. Projects often use components from other sources. You can use a special *Third-Party Codeline* in the Repository to manage this external code in a way that's similar to how we manage internally written code.

Testing, building, and integration

After you and your team members have done some coding, you need to incorporate these changes into the build. A central *Integration Build* accomplishes this by building from the codeline's latest state. Before you submit changes for the Integration Build, you must verify that your changes, when combined with everyone else's changes, will not break the build. You can do this by performing a *Private System Build*—a local version of the Integration Build incorporating your changes—in your workspace to ensure that the code accurately reflects the system that the Integration Build will create.

Testing is a complicated issue. Too much time spent testing means less time coding and a larger interval between when you make a change and when others can integrate it. Too little time spent testing means that you're more likely to introduce errors. Performing a *Smoke Test* before you check in your changes lets you ensure that that the system works well enough to share with others. We can rely on a Smoke Test to be enough for pre-check-in validation only if the Smoke Test is supported by a *Unit Test*, which lets us exhaustively test the component that we're working on and closely related components. To catch any errors that might slip through, we must also use a centrally run *Regression Test* that tests the system for known error conditions.

Does it matter?

Using these patterns together can give you a much improved and more agile development environment. You'll have more time to spend on coding and less on supporting activities. When used together, the practices support each other. Adding a single practice to your environment can help a little, adding practices that support it can help a lot, and missing one can hurt. In one instance, using the Active Development Line pattern improved the team's morale and productivity and the code's quality. Simplifying the pre-check-in testing worked well because the supporting practices (Unit Tests, Private Workspaces, and so on) were already in place.

Patterns and pattern languages help place solutions in context. Although many best practices exist, there is no one way to implement them, and sometimes best practices conflict. The SCM pattern language lets you establish basic SCM practices that help your team work effectively.

Steve Berczuk has been developing object-oriented software applications since 1989, often as part of geographically distributed teams. He's currently looking for a new opportunity. Contact him at steve@berczuk.com; www.berczuk.com.

Career Opportunities

DATA BASE ADMINISTRATOR:

Design database, code, test, implement. Establish database parametrs. Control access. Select codes of utility program for monitoring performance. B.S. req'd in comp. science/appl.math. 40 hrs./wk.; Job site L.A., Contact A. Sarafian (213) 624-6462. Fax (213) 624-3538

SUBMISSION DETAILS: Rates are $10.00 per line (ten lines minimum). Average five typeset words per line, eight lines per column inch. Add $10 for box number. Send copy at least one month prior to publication date to: Marian Anderson, Classified Advertising, *IEEE Software* Magazine, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1314; (714) 821-8380; fax (714) 821-4010. Email: manderson@computer.org.

Professors of Computer and Software Engineering and Chair of Software Engineering

Department of Computer Engineering

We are presently seeking candidates to fill several positions of Professor of Computer Engineering and Software Engineering, as well as a prestigious position of Chair of Software Engineering.

In addition to teaching at the undergraduate and graduate levels, successful candidates will work in one of the department's fields of research: software engineering, engineering large software systems, embedded computer systems, networking and mobile computing.

Successful candidates must have a Ph.D. in a related area and the academic background required to become a member of the *Ordre des ingénieurs du Québec*.

The new Department of Computer Engineering is rapidly expanding and presently includes over a thousand students. Researchers are provided with the necessary space and technical support, a modern computer environment and special funds to help them start up their research projects. For more information, visit the department website at www.gi.polymtl.ca.

Please send your curriculum vitae in French, your academic credentials, the names of three references and reprints of recent papers to:

Pierre N. Robillard, Ph.D., P. Eng., Head
Department of Computer Engineering
École Polytechnique de Montréal
P.O. Box 6079, Succursale Centre-ville
Montréal, Québec H3C 3A7 Canada
Email: pierre-n.robillard@polymtl.ca

ÉCOLE
POLYTECHNIQUE
MONTRÉAL

The application review process will begin immediately and will continue until the positions are filled.

According to Canadian Immigration requirements, Canadian citizens and permanent residents will be preferred. We are an equal-opportunity employer.