Software Configuration Management Patterns

BY **STEVE BERCZUK**

CONTENTS

ABOUT SCM PATTERNS

Software Configuration Management enables team members to work together more effectively. SCM touches all aspects of the development process, so SCM, done well, can improve engagement and productivity. Done poorly, SCM can slow a project down and cause frustration. An effective SCM process involves more than just the source code management system. Effective SCM requires that you think about the build and testing process, and also that you continually evaluate how modular your architecture is, and how effectively you communicate.

The Software Configuration Management patterns are most applicable to small teams that favor an agile software development approach, but they can help any team identify bottlenecks and work more effectively. If your team isn't agile, but wants to be, following these patterns will provide a framework for your team to develop more agile technical practices.

This Refcard describes some patterns that enable teams to work effectively. These patterns are described in detail in the book *Software Configuration Management Patterns: Effective Teamwork, Practical Integration.*

PATTERNS, PRACTICES, AND TOOLS

The patterns in this Refcard are tool-agnostic. Some tools support the practices implicitly, and some tools make it easier to implement the patterns, but any team can implement these patterns using any tool set. Think about how you work, first, then look into tools.

The kinds of tools you will want to have in place are:

- A Source Code Management System. Either a centralized SCM such as Subversion or a distributed one such as Git or Mercurial will work.
- A build tool that defines dependencies between components and allows for simple build and test execution. Gradle, Maven and Make are some examples.
- A Continuous Integration Server, for example, Jenkins, Go-CI, or Bamboo.
- A Documentation System. A Wiki or a well-known location in the SCM repository will do.
- An Artifact Repository, such as Artifactory, Nexus, or Archiva to keep built and third-party artifacts. A Maven repository manager or an SCM repository that can store binary files will work.

You will need to be able to:

- Add code to and check out code from an SCM repository.
- Commit changes to the SCM repository.
- Create a source code branch.
- Create a checkpoint (a tag or label) in the SCM repository.
- Monitor a code line for changes to trigger a build using the CI server.
- Build a project when changes are detected using the CI server.
- Alert the team when a build fails because of compile or test failures.

You can implement these patterns with any number of tools. Some tools realize the patterns implicitly.

THE PATTERNS

Rather than being isolated best practices, patterns are solutions to a problem in a context. When reviewing these patterns it is important to remember that applying single patterns will not be as effective as using a subset of related patterns together.

The SCM Pattern Language is based on the idea of delivering from a single code line and using as few other code lines as possible. Following these patterns will help your team to deliver code that is always ready to ship and minimize the time necessary to integrate new work into your application.

PATTERN	SUPPORTED BY
Main Line	Active Development Line
Active Devlopment Line	Private Workspace Private Versions Task Branch Release Line Release Prep Code Line Code Line Policy

PATTERN	SUPPORTED BY
Private Workspace	Third Party Code Line Repository Integration Build Privacy System Build
Repository	Third Party Code Line
Private Build	Smoke Test
Integration Buld	Task Level Commit Smoke Test
Third Party Code Line	
Task Level Commit	
Code Line Policy	
Smoke Test	Unit Test Regression Test
Unit Test	
Regressio Test	
Private Versions	Code Line Policy
Release Line	Code Line Policy
Release Prep Code Line	Code Line Policy
Task Branch	Code Line Policy

The SCM Patterns can help any team to be more productive, and they especially work well with agile technical practices such as unit testing, continuous integration, and continuous delivery.

The patterns can be grouped into 3 parts:

- Core patterns that the rest of the language supports.
- Workspace patterns that describe how developers work.
- Code line patterns that describe how to structure supporting code lines to enable delivery of new software from the main line.

CORE PATTERNS

The first decision you need to make is how to structure your code lines. The SCM Patterns describe how to work with fewer code lines, emphasizing integration over isolation.

MAIN LINE

The more code lines you have, the harder it is to understand the state of the project. While tools can help manage multiple code lines, the simplest way to minimize the overhead of branch management and context switching is to develop on a single Main Line.

The Main Line Pattern can help you to deliver rapidly with a team focused on a single project, or a set of related projects that deliver on the same cadence. With a Main Line pattern, all changes end up on a single stream of development. This provides the following advantages:

- Reduced merging and synchronization effort. Fewer code lines means fewer merges.
- More consistent integration, thus reducing the schedule risk of integrating late in the release cycle.
- If you are using the Main Line development model, strive to reduce branching to special situations such as:
- Releases. Create a Release Line to manage fixes to released code.
- Long-lived parallel efforts. Use a Task Branch for such efforts.
- Integration and Stabilization. A Release Prep Code Line has advantages over a code freeze for stabilization.

Since each branch is a potential distraction, limit branching to situations in which a branch increases stability and speed.

The risk with main line development is that developers might commit changes that break the code line. For development on a single code line to be effective, you need to make this code line a place where developers can feel confident that the code is working. An Active Development Line provides a framework for a more stable main line.

ACTIVE DEVELOPMENT LINE

Working on a Main Line only works when the code is always working. An Active Development Line is a Main Line where developers use practices to ensure that the code is in a working state, and that you can detect breakages as quickly as possible. An Active Development Line can support you when you need to do frequent releases and it is essential for agile software development. The patterns that support these practices are:

- Code Line Policy
- Unit Tests
- Integration Builds
- Private Workspaces
- Private Builds

To maintain the activity on the code line, define a Code Line policy that will keep serious defects out of the code line and integration build, but be willing to ignore trivial defects, as being too strict can paralyze the team.

The Code Line Policy for an Active Development Line could include rules like the following:

- The Build should run through the Private Build in a Private Workspace successfully before a commit.
- All changes will be accompanied by the appropriate tests, or a comment explaining why a test is not being performed.
- If the common Integration Build fails, the team will address it immediately.

The goal of the Active Development Line is to err on the side of progress, as an occasional broken build is less disruptive than merging larger changes that have not been integrated with the larger code base.

WORKSPACE PATTERNS

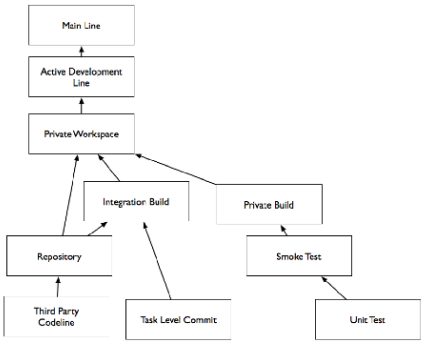

Figure 1: *Workspace Patterns*

PRIVATE WORKSPACE

To maintain an Active Development Line, you need an environment where developers can work with enough isolation that they are not disrupted, and identify integration issues before code is shared with the team. Developers need control over the state of the code they are working with so that they can work without distraction.

A Private Workspace is an environment where developers can build and test before accepting changes from the Active Development Line, or publishing them to the Active Development Line.

A Private Workspace has all of the dependencies a developer needs to work independently including:

- Code.
- The correct version of build tools.
- The correct version of dependencies.
- Configuration Files.

A developer needs to easily create the workspace from a simple set of instructions using the Repository, and a build tool that supports dependency management, such as Gradle or Maven. Tools such as virtual environments and containers can also help.

For the team to work effectively, developers need to follow the process in the Private Build pattern, updating their workspace frequently and committing changes frequently when the code is working to ensure that the status of the code accurately reflects the state of the Active Development Line.

REPOSITORY

To set up a new Private Workspace or an Integration Build you need to populate it from a repository that contains everything you need to build the code, including:

- Source Code
- Build Scripts
- Configuration
- Third Party Components

The Repository can be composed of a number of tools. The Source Code can be in a source code management system, components can be in an artifact repository, such as a Maven repository.

Ideally a developer should be able to create a workspace for a project in two steps:

- Check out a copy of the code from your SCM system.
- Run a build command (for example, a shell script or Gradle task) to configure and build the project.

The only documentation you should need is:

- The path to the project in the SCM System.
- The build command.
- Optional configuration changes to make for different environments.

You can document the workspace creation process in a well know location in the SCM repository, or on a CMS such as a wiki. A common convention would be to have a Getting Started page. Fewer, self-running scripts are better than a longer documented process, but the key attribute of a successful Getting Started process is that it can be executed without assistance from anyone else.

Having all dependencies in a single repository and simple procedure for creating a workspace will minimize the risk of introducing bugs that are related to environmental differences and improve efficiency when people join or move between projects.

PRIVATE BUILD

To avoid breaking the Active Development Line, perform a Private Build in your Private Workspace before committing changes. This will allow you to detect integration errors before they affect other developers.

The Private Build:

- Builds the code.
- Runs Smoke Tests.
- Runs Unit Tests.
- Creates a deployable artifact.

The Private Build should be identical to the Integration Build, or at least as close as possible. While IDEs can be used for day-to-day development, the build tools in your integration environment should be treated as the canonical build tool. The default private build may occasionally skip some tests in the interest of speed, but it should be possible to run the entire integration build (including tests) in a developer workspace.

To avoid checking in code that will break the integration Build developers run the Private Build as they develop. Before any commit developers should:

- Update their Private Workspace from the Active Development Line.
- Run the Private Build.
- Commit their changes only when the build passes.

The private build should be able to grab all dependencies automatically, and not rely on manual installs. A common mechanism for this is to pull dependencies from a Maven or Ivy repository using a build tool such as Gradle or Maven.

INTEGRATION BUILD

Building in a Private Workspace provides some assurance that all of the code works together, but you still want an automated mechanism to verify that the code that is in the version management system always builds and passes tests. An integration build runs automatically when changes are detected in the code line. The Integration Build:

- Updates the source in an integration workspace.
- Builds the code.
- Runs unit, smoke, and integration tests.
- Performs static analysis on the source code.

The integration build should be automated, fairly quick, and failures should be addressed immediately. If running a complete suite of tests takes too long, split the integration build into 2 phases, one which runs unit and smoke tests, and one which runs more thorough integration and regression tests.

THIRD PARTY CODE LINE

All of your locally developed code is in your Repository. Code from outside the organization that you depend on should also be in your Repository, as you need a way to manage dependencies. For binary dependencies, you can identify versions in your build configurations and use a repository manager. When you need to make customizations to open source code, you might want to manage the source code in your repository. A Third Party Code Line is a way you can easily manage local customizations to code.

- Add the third party source to your SCM repository.
- Label the original source.
- Create a branch for your local changes.
- When there is a new release of the third party code, add it to the mainline. Create a new branch for this code.
- Merge any relevant changes from the old branch to the new branch.

Once this is done, create an integration build for the code, and a mechanism for developers to reference the third party artifacts. If you are actively using an open source library, and your changes are not proprietary, consider contributing them to the project.

TASK LEVEL COMMIT

To help ensure that the Integration Build reflects the current state of the code, organize code changes by task oriented units of work and commit frequently. Associate each Task Level Commit with an issue from your issue tracking system to improve traceability. A Task Level commit is:

- Small. Commit changes when you have completed a unit of work.
- Frequent. Commit code as often as possible while maintaining working code.
- Associated with a feature being developed. For example, each commit could have an issue number mentioned.

You might commit after each of these steps:

- Add a method and a unit test.
- Use the new method.

Many issue tracking systems can associate commits with the issue identifiers, either by metadata or by finding issue IDs in the commit comments. Associating each commit with an issue is important to:

- Identify code changes that went into implementing an issue. This is useful for auditing and research.
- Identify the effort required for features.
- Help developers focus their efforts on useful features.

Be sure to update and build code before committing changes to the Main Line.

The Task Level commit is also a good pattern to follow when working on a Release Line.

SMOKE TEST

An Integration Build and Private Build use testing to help ensure that your code line is an Active Development Line. To verify that the code line still works after a change, run a Smoke Test after each change as part of the build. A smoke test is a quick to run integration test which:

- Is fast.
- Is Self-Evaluating.
- Provides broad coverage.
- Is Runnable by developers as part of a build-time test.

Smoke tests do not replace all manual quality assurance and analysis efforts, or Regression Tests, but they do allow for a way to catch common, critical errors quickly after each change.

UNIT TEST

Smoke Tests provide a quick way to make sure that the application works at a high level. You can rely on smoke tests only if you also have a mechanism to verify that your modules still work after you make a change. Unit tests are tests that test low level APIs and contracts.

Unit Tests are:

- Automated.
- Self-Evaluating.
- Fine Grained.
- Isolated. A unit test does not interact with other tests.

Unit tests test the contract that a class has with other components. Run unit tests while you are coding, before you check in changes, and as part of the build.

Writing unit tests as you code will also help you to identify coupling between modules so that you can remove it if it is inappropriate. Applying practices such as Test Driven Development, where you write tests before you write code, can be one way to ensure that you have good test coverage.

Unit tests can also help to identify module level regressions If the existing tests, and the tests that went with the change you are merging, both pass, you can be more confident that you merged the changes correctly.

Using a framework like xUnit can simplify your unit testing process.

REGRESSION TEST

Unit tests and smoke tests are designed to be fast and are meant to be run frequently. You still need a more comprehensive way to ensure that existing code does not get worse as you make other improvements.

Regression tests are typically a kind of integration test, though you can also write more isolate tests – more akin to unit tests - to detect regressions. Regression tests are often driven by problems that you found reactively, and might take longer to run than a build time test should. Ideally, regression tests will be automated.

Regression tests should cover:

- Problems you find in the QA process.
- User-reported problems.
- System level requirements.

When you find an error in a released build, it's a good practice to add a test that identifies the issue to the build.

If the Regression Tests don't take too long to run, add them to the main integration build. Otherwise run them as a second stage build, and consider adding "run regression tests on build" to the code line policy of a Release Line.

CODE LINE PATTERNS

While a single main line that always builds is ideal, there are times when you may want to create branches to make it easier to keep the main line stable and active. The code line patterns describe the types of code lines as well as the concept of a code line policy.

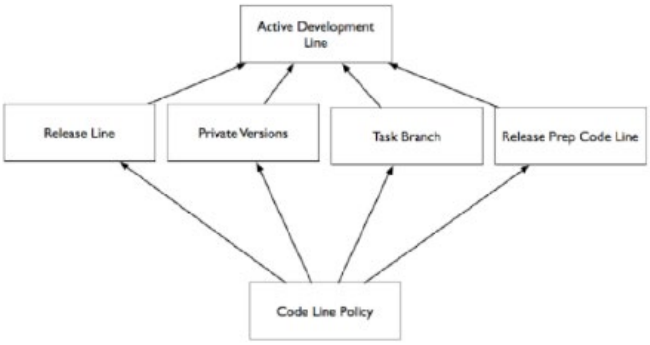

Figure 2: *Code Line Patterns*

RELEASE LINE

Active Development Line development is a simple and powerful approach to efficiently developing software. But developing software in a way that you can deliver at any point in time is challenging. It is useful to have the ability to maintain released versions without interfering with your current development by creating a Release Line. A Release Line is probably one of the more commonly used patterns.

As you release code to customers, identify the version of the release by using a tag, and create a branch at that tag. You can then make changes to that branch should you need to deliver a fix.

When appropriate, integrate changes from the Release Line into the Main Line, either by a merge, or, if the code has diverged significantly, by a functionally equivalent change.

A Code Line Policy for a Release Line might include:

- Each change should be in response to a documented issue.
- Each commit must reference an issue identifier.
- Changes should be reviewed with another team member before commits.
- Any change should be accompanied by a change to an existing test, a new test, or a reason why there is no test.
- Before committing a change, a developer should run additional tests, such as regression tests, in their Private Workspace.
- If the Integration Build fails, every team member stops to help address the issue.

The details are team dependent; the goal is to have a very stable Release Line.

In some ways, development on a Release Line is similar to a Waterfall-like process, and as such Release Lines should be used with care. A Release Line should be updated rarely, and only in the event of a critical release because each parallel work stream that you need to support can be a distraction.

RELEASE PREP CODE LINE

The goal of Active Development Line development is to release products from the tip of the code line. In some cases, teams still need extra time to stabilize code before a release, while also enabling new work. The traditional approach to this, a code freeze, has a number of down sides, including the opportunity for idle time and/or teams doing work without committing changes to the Main Line.

You can provide for a stabilization period and avoid the down-sides of a code freeze by doing the stabilization work on a Release Prep Code Line.

Create a branch when the majority of the team is ready to start work on the next release, and the current release is feature complete. This branch will accept changes from everyone on the team.

If you have a good set of automated tests in place, you will not need to use this pattern. But it is an alternative to a code freeze for teams that still require an extended integration test period.

A Release Prep Code Line is a middle ground between an Active Development Line and a Release Line, so it should have a Code Line Policy that emphasizes that changes should be small and tactical. For example, the policy for a Release Prep Code Line might not require the same degree of pre-commit validation as a Release Line commit, but a similar degree of validation. An example policy could be:

- All changes should be in response to a bug reported against an issue scheduled for release.
- Each commit should identify the issue number it addresses.
- Before code is committed, it should be buddy reviewed.
- Follow the same pre-commit build and test process as for an Active Development Line.

Code freezes should usually be avoided, and Release Prep Code Lines are a stop gap until your team is at a point where it can release code without a code freeze or a long stabilization cycle.

TASK BRANCH

While there are advantages to developing on an Active Development Line there are times when it's useful to be able to do work in parallel, isolated from the rest of main development work. Another name for a Task Branch is a Feature Branch. Use a task branch when:

- A subset of your team needs to collaborate on a speculative long-lived task that is a divergence from the main line. In this case, this branch will be shared among the team.
- When you are ready to start work on a feature for the next release before the current release is done.
- As part of a routine code review/pull request flow. In this use case, strive to integrate the Task Branch with the Main Line rapidly, typically within a day of creating it.

Because task branches delay integration, use them rarely and only when the benefit outweighs the overhead of the branch. When using a Task Branch, merge changes from the Main Line frequently so that you are aware of potential conflicts. A Task Branch ends in one of the following ways:

- It's abandoned.
- It is merged with the main line.

When working on a Task Branch, it is important to merge changes from the mainline into the Task Branch frequently to identify potential conflicts with other work. At the end of the Task Branch, the changes are merged into the main line.

PRIVATE VERSIONS

A source code management system's primary purpose is to facilitate collaboration among team members. The facilities it provides to checkpoint steps along the way to implementing a feature make it easy to recover from a mistake, and help make team members more willing to try things. Since you don't want to check in changes to the Active Development Line before code is in a consistent, working state, you want to be able to experiment with a complex change locally while still being able to take advantage of features of a version control system.

You can implement Private Versioning either by creating a private branch in the team repository, using a private repository, or by taking advantage of the local history feature of an IDE. If you are using a DVCS such as Git, local commits that you don't push to the central repository serve the same purpose.

Use a Private Versioning mechanism when you are making a significant change that you don't want to share with the team until it is complete, or when you are investigating options and want to checkpoint working increments. Be careful to limit the amount of time you are working on a private version, as working on a private version defers integration and thus increases risk. And integrate the current state of the Main Line with your workspace frequently.

CODE LINE POLICY

At the core of the SCM Patterns are practices that developers follow when working on code from a code line. To make developers aware of rules for a code line, create a Code Line policy to help developers decide what procedures to follow before committing changes to a code line. If possible, automate

enforcement of these policies. The code line policy identifies differences between code lines. A code line policy specifies:

- The reason for the code line (for example, active development, fixes for released code, a Task Branch).
- Rules to follow before committing changes (test, code review, etc.).
- Whether the Code Line is long-lived or transient.
- Access restrictions for various roles/individuals/groups.

For example:

- An Active Development Line might have a policy that requires Issue Numbers for every commit, and smoke and unit tests be run before the commit.
- A Release Line might also require that changes be reviewed before being committed, and that all commits should have an associated automated test.

You can enforce most of these rules by using tools such as build time steps or SCM triggers. The policies define the agreement between team members about how the code line works.

GENERAL GUIDELINES

This section describes some guidelines to consider when implementing the patterns. The SCM Patterns are a central part of your development process, and can highlight gaps in your process.

PRINCIPLES

When using the SCM patterns, keep the following principles in mind:

- Fewer code lines help you to focus on delivering customer value.
- Testing is as important to successful release management as version control.
- Integrate early and often to identify potential problems as early as possible and minimize schedule risk and wasted effort.
- When you want to create a branch, consider whether or not there is a simpler solution.

CAVEATS

A single Active Development Line is the best way to achieve the goals of frequent delivery. The Release Prep Code Line and Task Branch are adaptive patterns that provide a way for a team to make progress and they help to maintain an Active Development Line, but are workarounds to problems you may have with maintaining stability. Use these patterns sparingly with an eye towards being able to deliver from a single code line.

TESTING

Testing is essential to having the SCM Patterns work. Tests validate that the changes in a developer's workspace will not disrupt other developers unexpectedly when they are committed to the Main Line, and also provide a second level of assurance that the integration build works as expected. Though not normally considered part of Software Configuration Management, tests are the best way to validate that the "software configuration" on the code line is correct.

BUDDY REVIEWS AND PAIR PROGRAMMING

Rather than formal code reviews, teams can benefit from collaborating with a team member before sharing their work more widely. A buddy review process in which a developer asks another developer (the "check-in-buddy") to listen while they explain the code changes they are about to check in. While the check-in buddy can sometimes provide suggestions, the main benefit of this process is that the process of having to explain what you did can force critical thinking and help you to identify problems. Pair Programming is a more structured approach that provides a higher level of collaboration and shared understanding, but it is harder to do.

DEVOPS AND CONTINUOUS DELIVERY

The SCM Patterns can support DevOps practices by encouraging Private Build and test environments that transition cleanly to production-like environments.

BALANCE

The goal of a team using the Active Development Line model is to keep the code line working. Teams should take reasonable care to ensure that developers don't check in broken code. But mistakes happen, and the patterns provide for layers or checking. A developer might forget to update code before building and testing, and an error may make it into the integration build. The goal of the patterns is early detection of errors, not absolute prevention of them. Erring on the side of avoiding errors can also stop progress.

RESOURCES

The SCM Patterns describe approaches to development that are mostly independent of tools, though the right tools will make the process simpler. This approach also relies on a good understanding of techniques such as unit testing and continuous integration.

CONTINUOUS INTEGRATION PRACTICES

- Refcard on Continuous Integration Practices: refcardz.dzone.com/refcardz/continuous-integration
- Refcard on Continuous Integration Servers and tools: refcardz.dzone.com/refcardz/continuous-integration-servers

CONTINUOUS INTEGRATION TOOLS

- Jenkins: jenkins.io
- Bamboo: atlassian.com/software/bamboo

- Hudson: hudson-ci.org
- GoCD: www.gocd.org

REPOSITORY MANAGEMENT TOOLS

- Artifactory: jfrog.com/artifactory
- Nexus: sonatype.com/nexus-repository-sonatype

BUILD TOOLS

- Gradle: gradle.org
- Maven: maven.apache.org
- Ant: ant.apache.org
- Buildr: buildr.apache.org

DEVELOPMENT PRACTICES

- Test Driven Development Resources: testdriven.com
- DevOps Resources: devops.com
- Continuous Delivery: continuousdelivery.com
- Trunk Based Development: trunkbaseddevelopment.com

UNIT TESTING

- Refcard on Java Unit Testing tools: refcardz.dzone.com/refcardz/junit-and-easymock
- The JUnit web site has links to other Unit Testing frameworks: junit.org
- Guidelines on Unit Testing: extremeprogramming.org/rules/unittests.html
- Guidelines on Test Driven Development: testdriven.com

SCM TOOLS

- Git: git-scm.com
- Refcard on Git: refcardz.dzone.com/refcardz/getting-started-git
- Subversion: subversion.apache.org

CONCLUSION

Maintaining a single working code line sounds simple in theory, but has many practical challenges. The advantages such an approach offers in terms of simplicity and minimizing integration costs across the team can make the cost worthwhile. Doing Software Configuration Management effectively involves more than just SCM tools and code line management, but also your approach to development, including testing.

ABOUT THE AUTHOR

STEVE BERCZUK is a software developer who is passionate about helping teams deliver effectively by technical, process, and people management. A Scrum Master and Agile software configuration management expert, Steve is a frequent contributor to Techwell (www.techwell.com) and the author of the book "Software Configuration Management Patterns: Effective Teamwork, Practical Integration." Steve's articles and his blog can be found at www.berczuk.com, and you can follow him on Twitter @sberczuk.

Refcardz Feedback Welcome: refcardz@dzone.com

Sponsorship Opportunities: sales@dzone.com

DZone, Inc. 150 Preston Executive Dr. Cary, NC 27513

888.678.0399 - 919.678.0300