SD BEST PRACTICES 2007 CONFERENCE & EXPO

September 18-21, 2007
Hynes Convention Center
Boston, MA

Effective SCM: Better Feedback, Higher Quality and Happier Stakeholders

Steve Berczuk
Sr Software Engineer
Aveksa, Inc
Waltham, MA
steve@berczuk.com

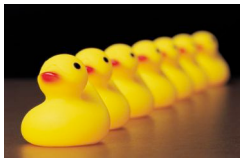
the future of software development

Agenda

- SCM and the Development Process.
- Common Problems
- SCM Concepts
- Solutions and Patterns for Better SCM
- Questions

What is SCM?

- Software Configuration Management
- Version Management
- Configuration Identification
- Anything Else?



Traditional View of SCM

- Configuration Identification
- Configuration Control
- Status Accounting
- Audit & Review
- Build Management
- Process Management, etc

Agile/Effective SCM

- Who?
- What?
- When?
- Where?
- Why?
- How?

Focus on processes that add value.

Why Do We Do SCM?

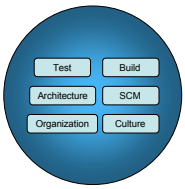
- Control.
- Adaptability.
- Robustness.
- Identification.
- ??

Who Does SCM?

- Release Engineers.
- Developers.
- Customers?
- ??

SCM and the Big Picture

- Architecture
- Culture/Organization
- Build
- Test
- SCM
  - Version Management



Common Problems (I)

- Not Enough Process
  - "Builds for me!"
  - "Works for me…!"
  - "The build is broken again!"
  - Ad-hoc code sharing

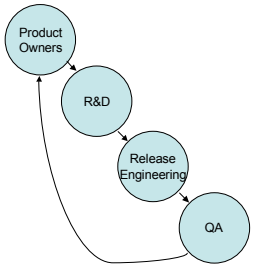
Common Problems (II)

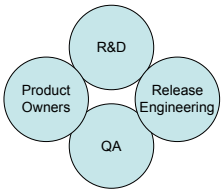
- Process Gets in the Way.
  - Pre-check-in testing takes too long
  - Code freeze/idle resources
- Long Integration Times at Project Release.
  - "Fixing it" in integration

Agility and SCM

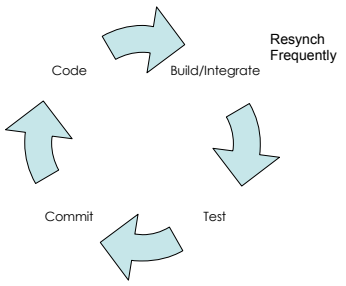
- Agile Methods emphasize:
  - Feedback
  - Communication
  - Process that adds value
- Agile SCM
  - Effective SCM
  - Simple SCM
  - Not only for Agile teams

Feedback and The Team



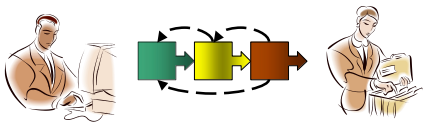


The Agile SCM Cycle



Continuous Refinement

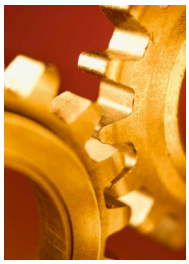
- At each phase do as much as needed.
  - But no more
- Feedback to decide when to add processes.
  - But only if it adds value



Developer Customer

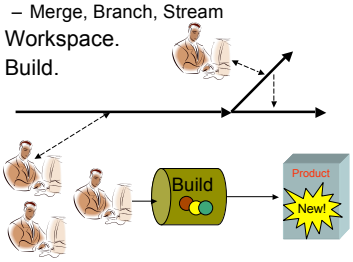
Themes

- Automate where possible.
- Simple steps.
- Reproducible steps.
- “Fractal” processes.



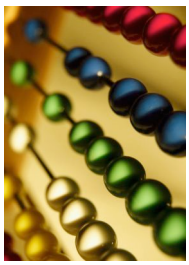
Concepts

- Codeline.
  - Merge, Branch, Stream
- Workspace.
- Build.



Enough Control For The Task

- Codelines.
- Policies.
- Enforcement.
- Automation.



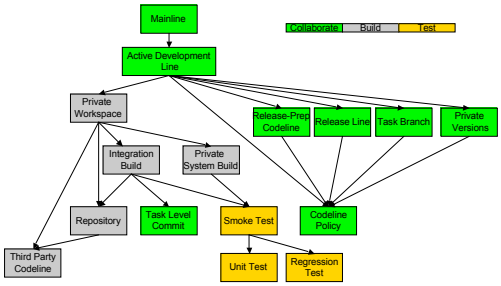
Workspace

- Everything you need to build an application:
  - Code
  - Scripts
  - Database resources, etc

Creating an Agile SCM Environment

- Decide on a goal.
- Choose an appropriate Codeline Structure.
  - Establish the related policy.
- Create a process to set up workspaces.
  - Private
  - Integration
- Build & Deploy is an Iteration 0 Story.
- Integrate frequently at all levels.
  - Developer Workspace
  - Integration Build
- Deploy frequently.
- Test.

The SCM Pattern Language

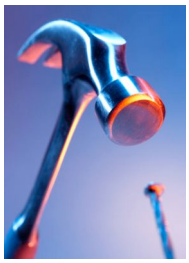


Today

- Codelines
- Workspaces
- Build
- Test

Tools

- Tools help.
- Tools should support your process.
  - Don't pick tool first.
- Follow tool's paradigm.
  - Once you are committed.



Codeline Structure Issues

- How many codelines should you be working from?
- What should the rules be for check-ins?
- Codelines are the integration point for everyone's work.
- Codeline structure determines the rhythm of the project.

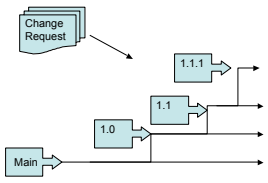
Mainline

- You want to simplify your codeline structure.
- **How do you keep the number of codelines manageable (and minimize merging)?**

Mainline (Forces & Tradeoffs)

- A Branch is a tool for isolating work.
  - Branching can require merging.
  - Merging can be difficult.
- Codelines are a logical way to organize work.
- You will need to integrate everything eventually.
- You want to maximize concurrency.
- You want to minimize problems cause by deferred integration.

Too Many Branches?



Or the right number?

Mainline (Solution)

- When in doubt, do all of your work off of a single *Mainline*.
- Understand why you want to branch, and consider the costs.
- Unresolved:
  - Simplicity with speed and *enough* stability: *Active Development Line*

Active Development Line

- You are developing on a *Mainline*.
- **How do you keep a rapidly evolving codeline stable enough to be useful (but not impede progress)?**

Active Development Line (Forces)

- A Mainline is a synchronization point.
- More frequent check-ins are good.
- A bad check-in affects everyone.
- If testing takes too long: Fewer check-ins:
  - Human Nature
  - Time
- Fewer check-ins imply a slower rhythm.

Active Development Line

- Use an Active Development Line.
- Have “good enough” check-in policies for.
  - More structure where needed.
- Establish practices for an active codeline:
  - Doing development: Private Workspace
  - Keeping the codeline stable: Smoke Test
  - Managing maintenance versions: Release Line
  - Dealing with potentially tricky changes: Task Branch
  - Avoiding code freeze: Release Prep Codeline

Private Workspace

- You want to support an *Active Development Line*.
- **How do you keep current with a dynamic codeline and also make progress without being distracted by your environment changing from beneath you?**

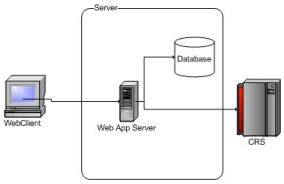
Private Workspace (Forces)

- Frequent integration avoids working with old code.
- People work in discrete steps: Integration can never be “continuous.”
- Sometimes you need different code.
- Excessive isolation makes life difficult for all.

Private Workspace (Solution)

- Create a *Private Workspace* that contains everything you need to build a working system.
  - You control when you get updates.
  - You can test before committing changes.
- Before integrating your changes:
  - Update your workspace.
  - Build your workspace.
  - Test your code and the system.

Private Workspace Example



- Workspace
  - App Server
  - Database Schema
  - Code for Web App
  - Test CRS Login
  - (Build/Deploy and Configuration Tools & Scripts)

Private Workspace Requires

- Populate the workspace: *Repository*
- Manage external code: *Third Party Codeline*
- Build and test your code: *Private System Build*
- Integrate your changes with others and test: *Integration Build*

Repository

- *Private Workspace* and *Integration Build* need components.
- **How do you get the right versions of the right components into a new workspace?**

Repository (Forces & Tradeoffs)

- Many things make up a workspace
  - code, libraries, scripts.
- You want to be able to easily build a workspace from nothing.
  - New developers
  - Integration workspaces
- Components could come from a variety of sources (3rd Parties, other groups, etc).
- Reproducibility

Repository (Solution)

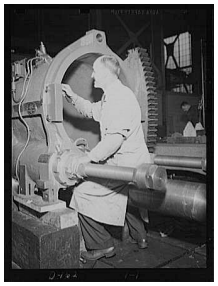
- Have a single point of access for everything.
- Have a mechanism to support easily getting things from the *Repository*.
  - Install Version Manager Client
  - Get Project from Version Management
  - Build, Deploy, Configure (Ant target, Maven goal)
  - Simple, repeatable process.
- Unresolved:
  - Manage external components: *Third Party Codeline*

Examples

- “Getting Started” wiki or web page.
- Maven script (scm:bootstrap)
- Ant script.
- Repository “check-out” using repository layout.

Task Level Commit

- You need to associate changes with an *Integration Build*.
- **How much work should you do before checking in files?**

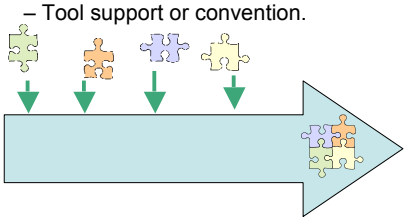


Task Level Commit (Forces)

- Smaller tasks: easier rollback.
- Larger commits: less testing time.
  - If using CI perhaps fewer builds.
- Many small issues: 1 check-in is tempting.
- Issue tracking systems
  - Identify units of work.
  - Release notes.

Task Level Commit (Solution)

- Do one commit per small-grained task.
- Associate changes with issues.
  - Tool support or convention.



Types of Tests

Common Name	Author	Created	Isolation	Purpose
Unit/Programmer	Developer	During Unit Dev	High	Testing functional components
Smoke (Integration)	Developer QA	"Integration"	Low	Verify minimal operation.
Regression	Support QA Developer	Post Release	Low	Verify that problems do not resurface

Smoke Test

- You need to verify an *Integration Build* or a *Private System Build* so that you can maintain an *Active Development Line.*
- **How do you verify that the system still works after a change?**

Smoke Test (Forces)

- Exhaustive testing is best for ensuring quality.
- Longer tests imply longer check-ins.
  - Less frequent check-ins.
  - Baseline more likely to have moved forward.
- People have a need to move forward.
- Stakeholders have a need for quality and progress.
- (Automated) Test Execution Time is often idle time.

Smoke Test (Solution)

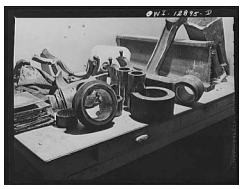
- Subject each change to a Smoke Test that verifies that the application has not broken in an obvious way.
  - Before a commit. (after Private System Build)
  - During Integration Build
- A Smoke Test is not comprehensive. You will need to find:
  - Problems you think are fixed: Regression Test
  - Low level accuracy of interfaces: Unit Test

Smoke Test Example

- Start up application
  - Seems trivial
  - Can ID issues with
    - Configuration
    - Packaging
    - Connectivity with databases

Unit Test

- A *Smoke Test* is not enough to verify that a module works at a low level.
- **How do you test whether a module still works after you make a change?**

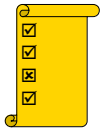


Unit Test (Forces)

- Integration identifies problems, but makes it harder to isolate problems.
- Low level testing is time consuming.
- After a change to a module things can break.
  - Check to see if the module still works before integration
  - You can isolate the problems.

Unit Test (Solution)

- Develop and run *Unit Tests*
- Almost nothing is too trivial to test.
- *Unit Tests* should be:
  - Automatic/Self-evaluating
  - Fine-grained
  - Isolated
  - Simple to run



- Also known as *Programmer Tests*

- J.B. Rainsberger

Regression Test

- A *Smoke Test* is good
  - Not comprehensive.
- **How do you ensure that existing code does not get worse after you make changes?**

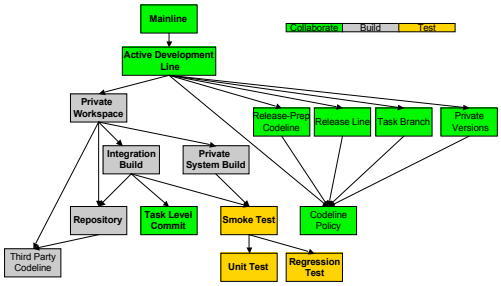
Regression Test (Forces)

- Comprehensive testing takes time.
- It is good practice to add a test whenever you find a problem.
- When an old problem recurs, you want to be able to identify when this happened.

Regression Test (Solution)

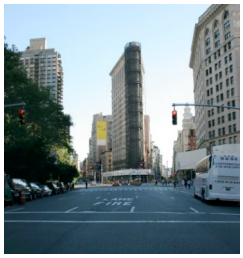
- Develop *Regression Tests* based on test cases that the system has failed in the past.
- Run *Regression Tests* whenever you want to validate the system.
- You can run these tests as part of an automated build (nightly or more frequently).

And Now



More than one codeline

- Stability
  - Releases
- Variations
  - Maintenance/Fixes
  - Customer Specific Changes
- Consider options
  - Branches sometime necessary.



Codeline Policy

- *Active Development Line* and *Release Line* (etc) need to have different rules.
- **How do developers know how and when to use each codeline?**

Codeline Policy (Forces)

- Different codelines:
  - Have different needs
  - Need different rules.
- People may not follow the rules.
- The rules need to make sense.
- How do you enforce/explain a policy?

Codeline Policy (Solution)

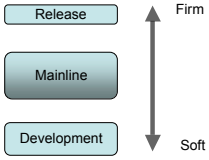
- Define the rules for each codeline as a *Codeline Policy*. The policy should be concise and auditable.
- Consider tools to enforce the policy.
- Consider branching on a policy change.

Sample Codeline Policies

- Active Development Line
- Release Line
- Other

Policies: The Tofu Scale

- Laura Wingerd (Perforce Software)
- Consider:
  - How close software is to being released.
  - How thoroughly must changes be reviewed and tested.
  - How much impact a change has on schedules.
  - How much a codeline is changing.
- See *Practical Perforce* for more info



Release Line

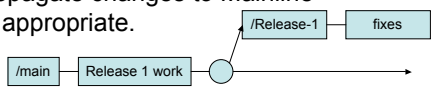
- You want to maintain an *Active Development Line*.
- **How do you do maintenance on a released version without interfering with current work?**

Release Line (Forces)

- A codeline for a released version needs a *Codeline Policy* that enforces stability.
- Day-to-day development will move too slowly if you are trying to always be ready to ship.

Release Line (Solution)

- Split maintenance/release activity from the *Active Development Line* and into a *Release Line*.
- Allow the line to progress on its own for fixes.
- Propagate changes to Mainline as appropriate.



Private System Build

- You need to build to test what is in your *Private Workspace*.
- **How do you verify that your changes do not break the system before you commit them to the *Repository*?**

Private System Build (Forces)

- Developer Workspaces have different requirements than the system integration workspace.
- The system build can be complicated.
- Committing changes that break the Integration Build is bad.
- It can be costly to fix mistakes after they are shared.

Private System Build (Solution)

- Build the system using the same mechanisms as the central integration build, a *Private System Build*.
  - Should match the integration build.
  - Do this before checking in changes!
  - Update to the codeline head before a build.
- Unresolved:
  - Testing what you built: *Smoke Test*

Integration Build

- What is done in a *Private Workspace* must be shared with the world.
- **How do you make sure that the code base always builds reliably?**

Integration Build (Forces)

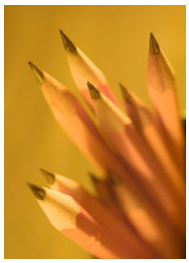
- People do work independently.
- *Private System Build*s are a way to check the build.
- Building everything may take a long time.
- You want to ensure that what is checked-in works.

Integration Build (Solution)

- Do a centralized build for the entire code base.
  - Use automated tools: Cruise Control, SCM tool Triggers, etc.
- Still Unresolved:
  - Testing that the product still works: *Smoke Test.*
  - Make build products available for clients in a *Repository.*
  - Figure out what broke a build: *Task Level Commit.*

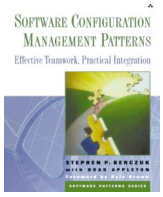
To Do List

- Long Term
  - Architectural Changes (configuration, modules)
- Medium Term
  - Automated Testing
- Short Term
  - Workspaces
  - Continuous Integration



Resources/Places to Go

- www.scmpatterns.com
- www.berczuk.com
- www.cmcrossroads.com
- steve@berczuk.com
- *Software Configuration Management Patterns:Effective Teamwork, Practical Integration*



Other Books of Interest

- *Pragmatic Version Control using Subversion* (Mike Mason)
- *Pragmatic Version Control using CVS* (Hunt and Thomas)
- *Practical Perforce* (Wingerd)
- *Pragmatic Project Automation* (Mike Clark)
- *Junit Recipes* (Rainsberger)
- *Release It!*

Questions?

ISBN: 0201741172