Software Configuration Management in Context:

Effective Teamwork, Practical Integration

December 8, 2004
Steve Berczuk

Agenda

- Background
  - SCM and The Development Process.
  - Patterns and SCM Pattern Languages.
  - Software Configuration Management Concepts.
- SCM Patterns
- Questions

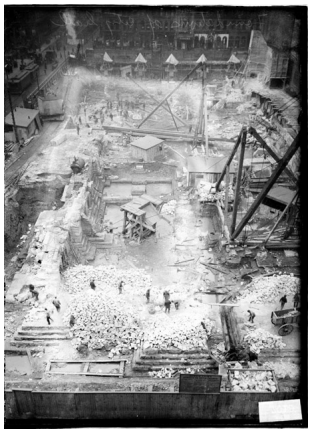
Goals

- Discuss Some Common Problems
- Learn how taking a “Big Picture View” of SCM will you make your process more effective.
- Understand how working with an Active Development Line Model Simplifies your process.
- See how to apply the SCM Pattern Language to help you to do this.

About Me

- Software Developer, Architect, Consultant, Author
- Startup and established company experience
- Systems ranging from Travel Web sites, to enterprise systems, to space science systems.
- Agile and Iterative Development.

Part I: Background/Foundation



Common Problems

- “Builds for me…”
- “Works for me!”
- Pre-check-in testing takes too long.
- The Build is Broken Again!
- Code Freezes.
- “What branch do I work off of?”
- Long integration times at end of project.

What is *Agile SCM?*

- *Individuals and Interactions* over Processes and Tools
  - SCM Tools should support the way that you work, not the other way around.
- *Working Software* over Comprehensive Documentation
  - SCM can automate development policies & processes: Executable Knowledge over Documented Knowledge.

...What is *Agile SCM*?

- *Customer Collaboration* over Contract Negotiation.
  - SCM should facilitate communication among stakeholders and help manage expectations.
- *Responding to Change* over Following a Plan.
  - SCM is about facilitating change, not preventing it.

Traditional View of SCM

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

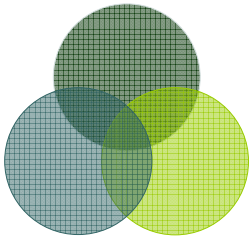
Effective SCM

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

Think about the entire value chain.

Part of the Puzzle

- Architecture
- Software Configuration Management
- Culture/Organization



The Goal: Working software that delivers value.

SCM as an Enabling Tool

- SCM Gives You:
  - Reproducibility
  - Integrity
  - Consistency
  - Coordination
- SCM Enables:
  - Increased productivity
  - Enhanced responsiveness to customers
  - Increased quality

SCM Done Badly Can:

- Slow down development
- Frustrate developers
- Limit customer options

Alternate Definition of SCM

- SCM is a set of structures and actions that enable you to build systems in repeatable, agile fashion while improving quality and helping your customers feel more confident.
- SCM facilitates frequent feedback on build quality and product suitability.

Core SCM Practices

- Frequent feedback on build quality, and product suitability
- Version Management
- Release Management
- Build Management
- Unit & Regression Testing

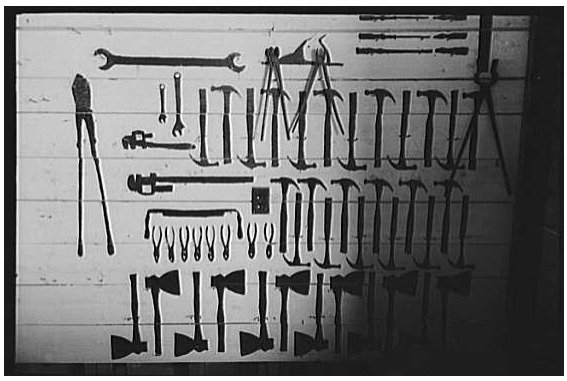
Effective Codeline Structures

- 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.

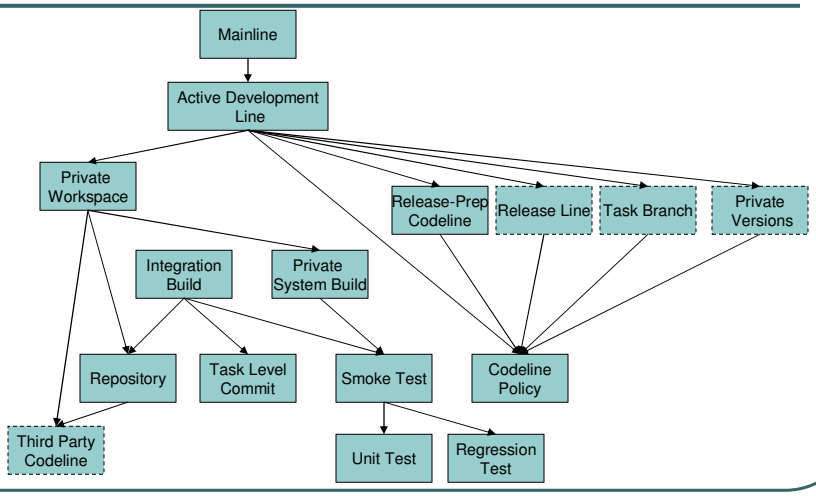
What are *Patterns* and Pattern Languages?

- A *pattern* is a solution to a problem in a context.
- Patterns capture common knowledge.
- Pattern *languages* guide you in the process of building something using patterns. Each pattern is applied in the correct way at the correct time.

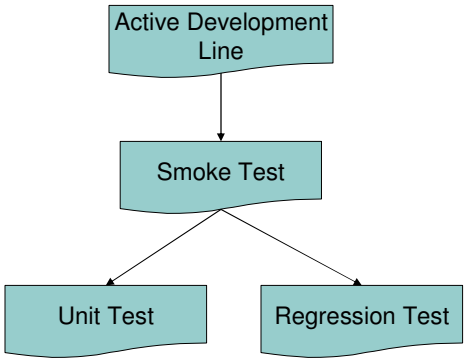
Part II: The Patterns



The SCM Pattern Language



A Word about Context



- *Smoke Test* "completes" *Active Development Line.*
- *Smoke Test* applies in the context of *Active Development Line.*
- Arrows point from context to the "next" pattern.

Mainline

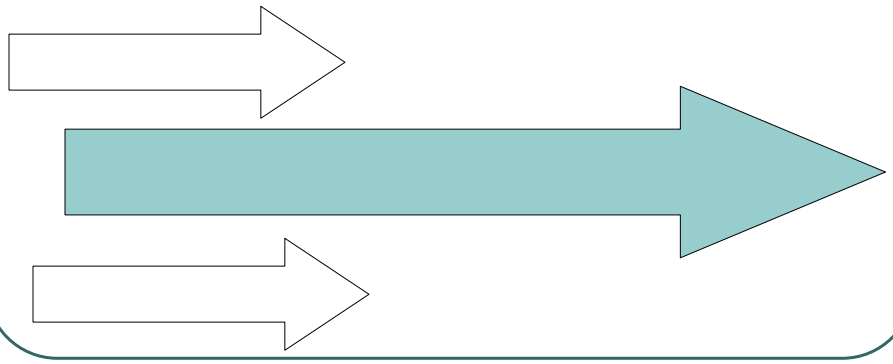
- You want to simplify your codeline structure.
- **How do you keep the number of codelines manageable (and minimize merging)?**

Mainline (Forces & Tradeoffs)

- A Branch is a useful tool for isolating yourself from change.
- Branching can require merging, which can be difficult.
- Separate codelines seem like a logical way to organize work.
- You will need to integrate all of the work together.
- You want to maximize concurrency while minimizing problems cause by deferred integration.

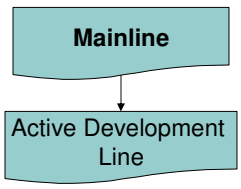
Mainline (Solution)

- When in doubt, do all of your work off of a single *Mainline*.



Mainline (Unresolved)

- Simplicity with speed and *enough* stability: *Active Development Line*.



Active Development Line

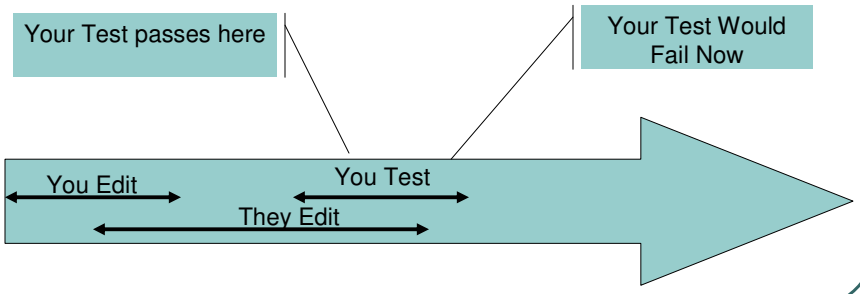
- You are developing on a *Mainline*.
- **How do you keep a rapidly evolving codeline stable enough to be useful (but not impede progress)?**

Active Development Line (Forces & Tradeoffs)

- 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 slow project's pulse.

Phase Shift

- Long running tests increase the likelihood of phase shift.



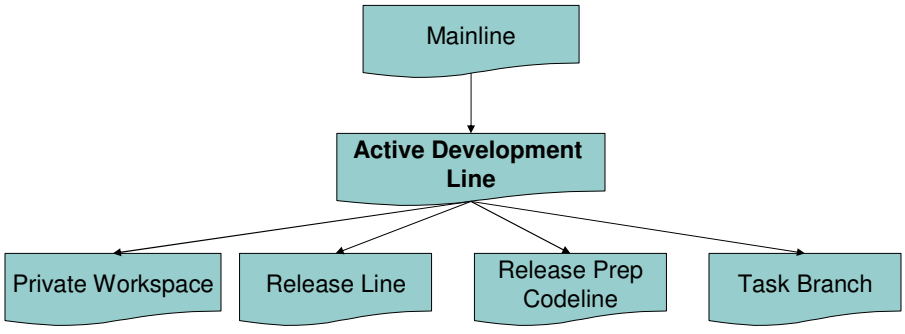
Active Development Line (Solution)

- Use an *Active Development Line*.
- Have check-in policies suitable for a "good enough" codeline.

Active Development Line (Unresolved)

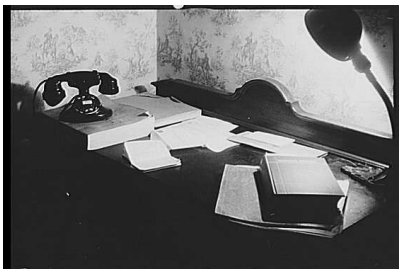
- Doing development: *Private Workspace*
- Keeping the codeline stable: *Smoke Test*
- Managing maintenance versions: *Release Line.*
- Dealing with potentially tricky changes: *Task Branch.*
- Avoiding code freeze: *Release Prep Codeline.*

Active Development Line Context



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 & Tradeoffs)

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

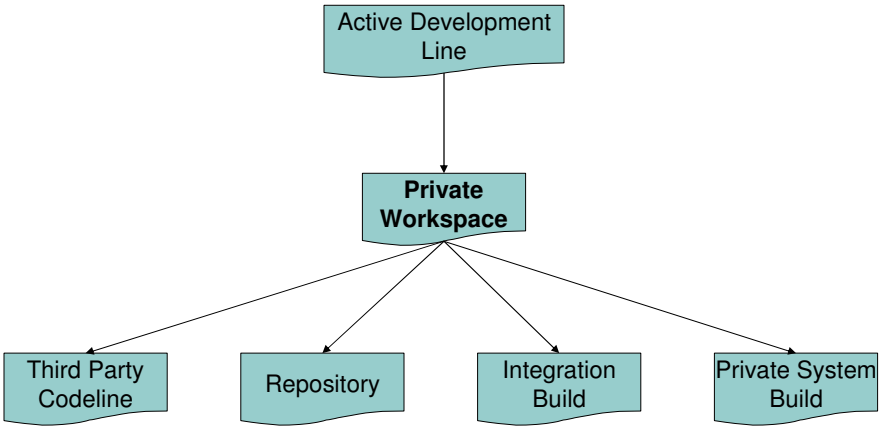
Private Workspace (Solution)

- Create a *Private Workspace* that contains everything you need to build a working system. You control when you get updates.
- Before integrating your changes:
  - Update
  - Build
  - Test

Private Workspace (Unresolved)

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

Private Workspace Context



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.
- These components could come from a variety of sources (3rd Parties, other groups, etc).

Repository (Solution)

- Have a single point of access for everything.
- Have a mechanism to support easily getting things from the *Repository*.

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 & Tradeoffs)

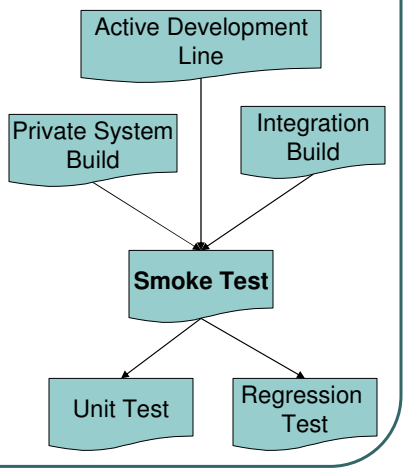
- Exhaustive testing is best for ensuring quality.
- The longer the test, the longer the check-in, resulting in:
  - Less frequent check-ins.
  - Baseline more likely to have moved forward.

Smoke Test (Solution)

- Subject each build to a *Smoke Test* that verifies that the application has not broken in an obvious way.

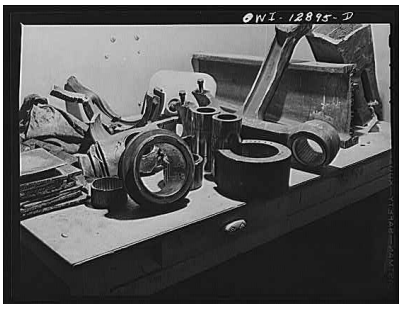
Smoke Test (Unresolved)

- 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*



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?**

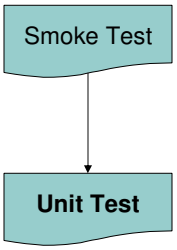


Unit Test (Forces & Tradeoffs)

- Integration identifies problems, but makes it harder to isolate problems.
- Low level testing is time consuming.
- When you make a change to a module you want to check to see if the module still works before integration so that you can isolate the problems.

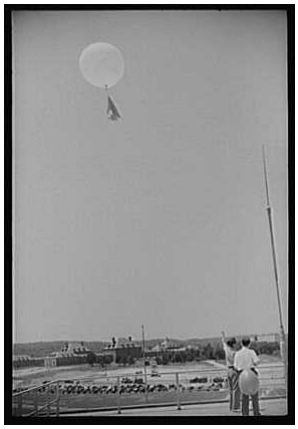
Unit Test (Solution)

- Develop and run *Unit Tests*
- *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 but not comprehensive.
- **How do you ensure that existing code does not get worse after you make changes?**

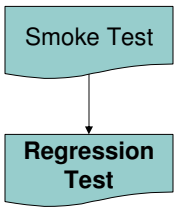


Regression Test (Forces & Tradeoffs)

- 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.



Release Line

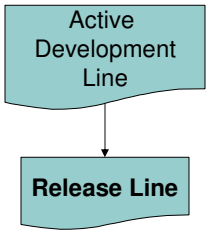
- You want to maintain an *Active Development Line*.
- **How do you do maintenance on a released version without interfering with current work?**

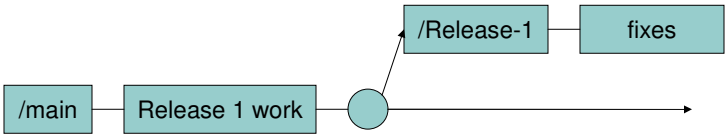
Release Line (Forces & Tradeoffs)

- 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)

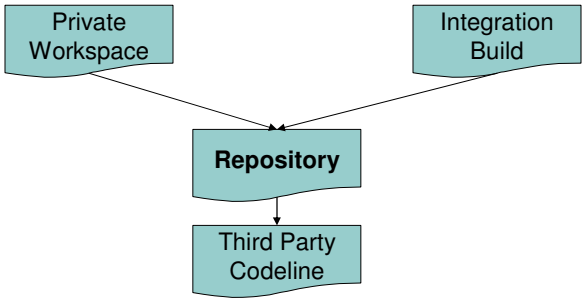
- Split maintenance/release activity from the *Active Development Line* and into a *Release Line*.
- Allow the line to progress on its own for fixes.





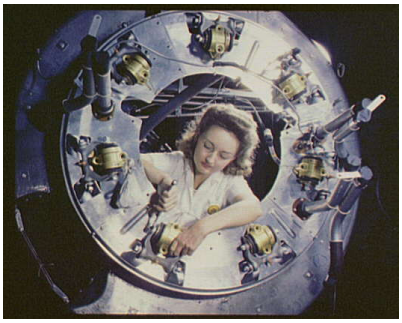
Repository (Unresolved)

- Manage external components: *Third Party Codeline*



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 & Tradeoffs)

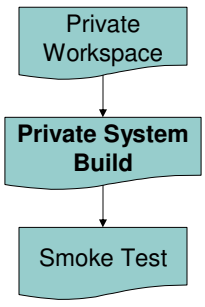
- Developer Workspaces have different requirements than the system integration workspace.
- The system build can be complicated.
- Checking things in that break the *Integration Build* is bad.

Private System Build (Solution)

- Build the system using the same mechanisms as the central integration build, a *Private System Build*.
- This mechanism should match the integration build.
- Do this before checking in changes!
- Update to the codeline head before a build.

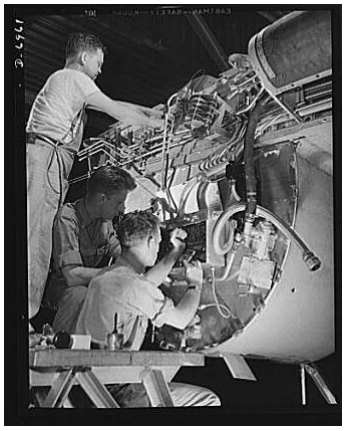
Private System Build (Unresolved)

- 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?**

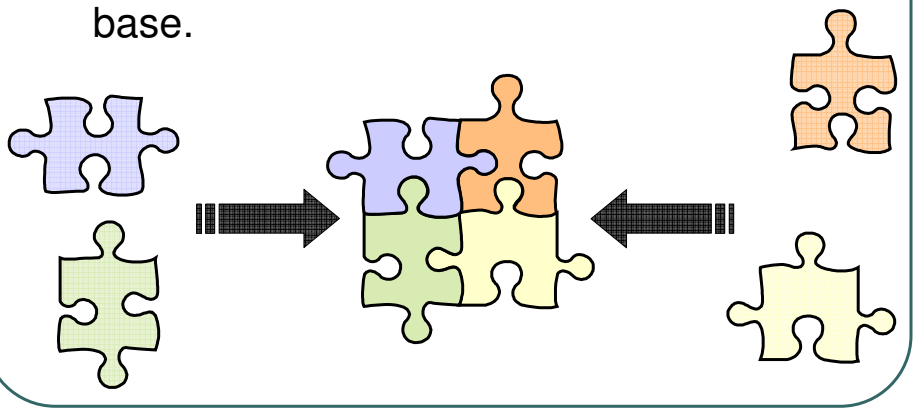


Integration Build (Forces & Tradeoffs)

- 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.

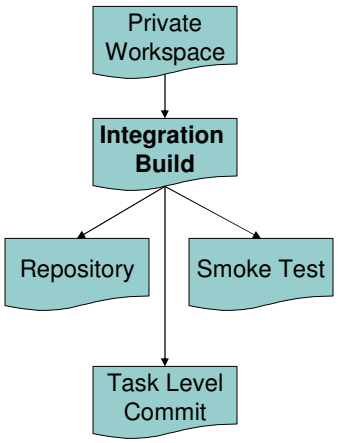
Integration Build (Solution)

- Do a centralized build for the entire code base.



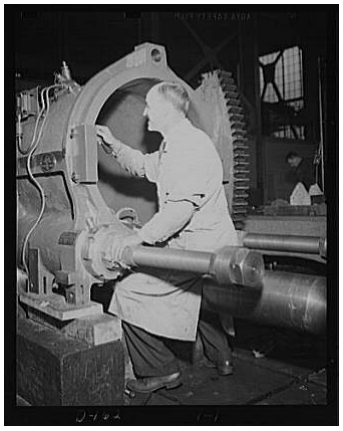
Integration Build (Unresolved)

- Testing that the product of the build still works: *Smoke Test*.
- Build products may need to be available for clients to check out.
- Figure out what broke a build: *Task Level Commit.*



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 & Tradeoffs)

- The smaller the task, the easier it is to roll back.
- A check-in requires some work.
- It is tempting to make many small changes per check-in.
- You may have an issue tracking system that identifies units of work.

Task Level Commit (Solution)

- Do one commit per small-grained task.

Codeline Policy

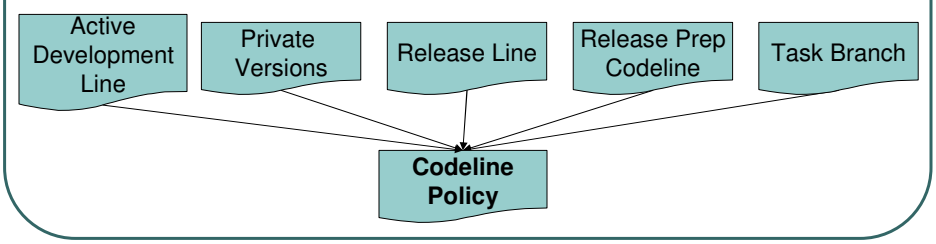
- *Active Development Line* and *Release Line* (etc) need to have different rules.
- **How do developers know how and when to use each codeline?**

Codeline Policy (Forces & Tradeoffs)

- Different codelines have different needs, and different rules.
- You need documentation. (But how much?)
- How do you explain a policy?

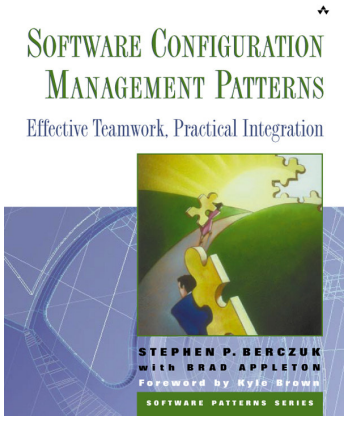
Codeline Policy (Solution)

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



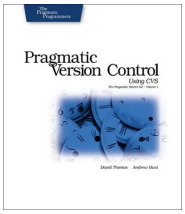
Wrap Up, Destinations

The SCM Patterns Book



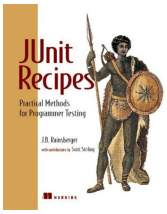
- Pub Nov 2002 By Addison-Wesley Professional.
- ISBN: 0201741172

Other Books of Interest



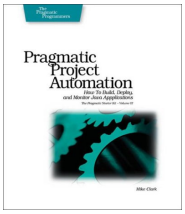
Pragmatic Version Control

by Andy Hunt & Dave Thomas



JUnit Recipies

by J. B. Rainsberger



Pragmatic Project Automation

by Mike Clark

Lean Thinking

- References:
  - ***Lean Software Development Toolkit***: Mary Poppendick and Tom Poppendick (2003). Addison Wesley.
  - ***Lean Thinking***: J Womack and D. T. Jones (2003). New York, Free Press.

Other Pointers

- www.scmpatterns.com
- acme.bradapp.net
- www.berczuk.com
- www.cmcrossroads.com

- steve@berczuk.com

Questions?