

# Patterns for Agile Software Configuration Management

Steve Berczuk

---

---

---

---

---

---

---

## Agenda

- Background
  - SCM and Agility.
  - Patterns and SCM Pattern Languages.
  - Software Configuration Management Concepts.
- SCM Patterns
- Questions

© 2003 Steve Berczuk   Patterns for Agile Software Configuration Management   2

---

---

---


---

---

---

---

## Part I: Background/Foundation



© 2003 Steve Berczuk   Patterns for Agile Software Configuration Management   3

---

---

---

---

---

---

---

# Patterns for Agile Software Configuration Management

## 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



---

---

---

---

---

---

---

# Patterns for Agile Software Configuration Management

## Agile SCM

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



---

---

---

---

---

---

---

---

## SCM as a Tool For Agility

- SCM Enables:
  - Increased productivity
  - Enhanced responsiveness to customers
  - Increased quality
- SCM Enables Agile Values
  - XP: Courage. You can reproduce things easily

---

---

---

---

---

---

---

---

## 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.

---

---

---

---

---

---

---

---

SCM Concepts



---

---

---

---

---

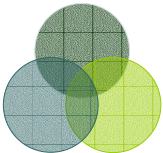
---

---

---

Part of the Puzzle

- Architecture
- Software Configuration Management
- Culture/Organization



---

---

---

---

---

---

---

---

What SCM Does for You

- Reproducibility
- Integrity
- Consistency
- Coordination

---

---

---

---

---

---

---

---

**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

---

---

---

---

---

---

---

# Patterns for Agile Software Configuration Management

## SCM Concepts & Definitions

- Codeline/Branch
- Versioning Concepts
  - Configuration
  - Version
  - Revision
  - Label
- Workspace

---

---

---

---

---

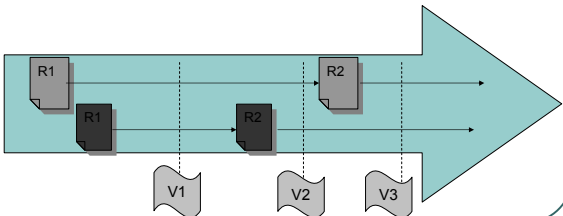
---

---

---

## Codeline

- A **codeline** contains every version of every artifact over one evolutionary path.



---

---

---

---

---

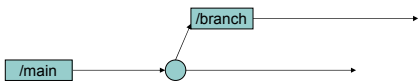
---

---

---

## Branching

- **Branch**: A codeline that contains work that derives (and diverges) from another codeline.
- **Branch** of a file: A revision of a file that uses the trunk revision as a starting point.



---

---

---

---

---

---

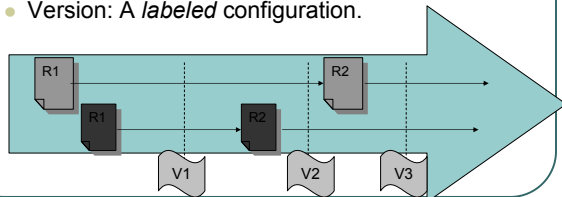
---

---

# Patterns for Agile Software Configuration Management

## Versions, Revisions and Labels

- Revision: An element at a point in time.
- Configuration: A snapshot of the codeline at a point in time.
- Version: A *labeled* configuration.



© 2003 Steve Berczuk Patterns for Agile Software Configuration Management 19

---

---

---

---

---

---

---

---

## Workspace

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



© 2003 Steve Berczuk Patterns for Agile Software Configuration Management 20

---

---

---

---

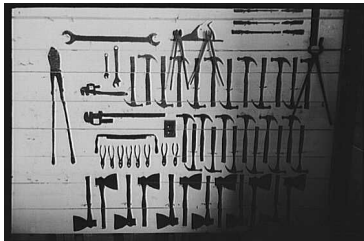
---

---

---

---

## Part II: The Patterns



© 2003 Steve Berczuk Patterns for Agile Software Configuration Management 21

---

---

---

---

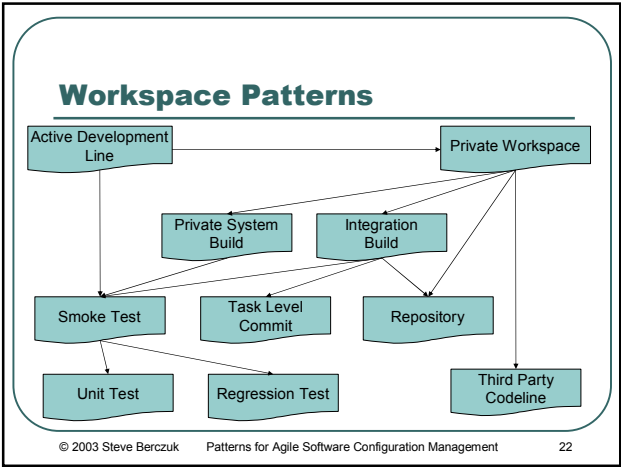
---

---

---

---

# Patterns for Agile Software Configuration Management



---

---

---

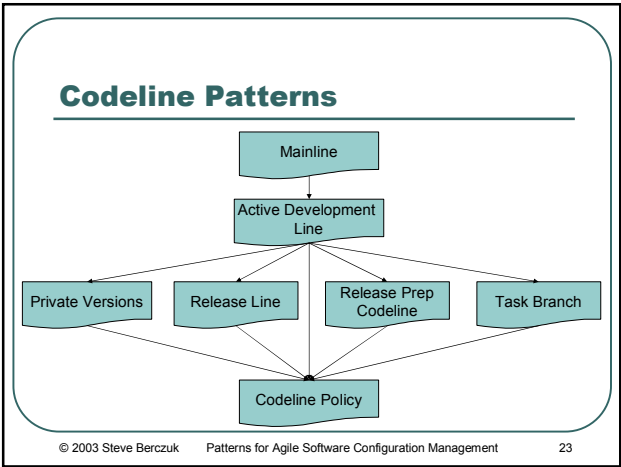
---

---

---

---

---



---

---

---

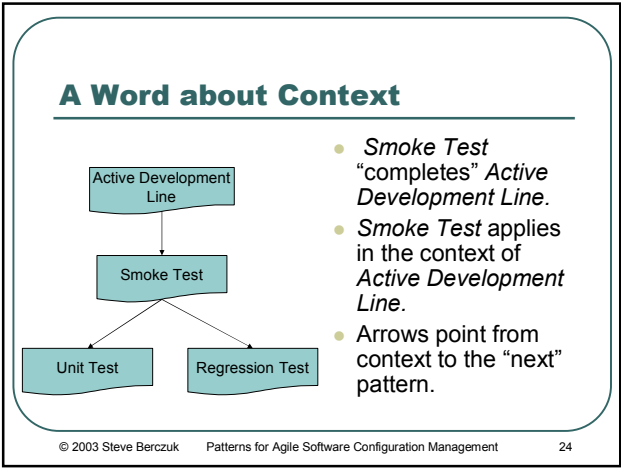
---

---

---

---

---



---

---

---

---

---

---

---

---

Agility and 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 pulse 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 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 caused by deferred integration.

---

---

---

---

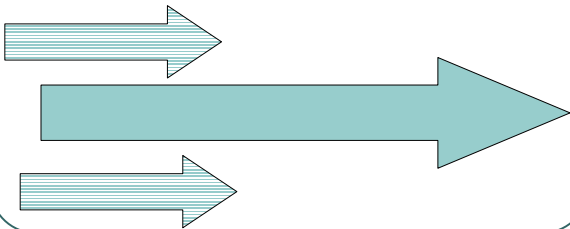
---

---

---

Mainline (Solution)

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



---

---

---

---

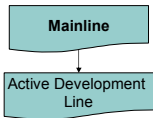
---

---

---

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.

---

---

---

---

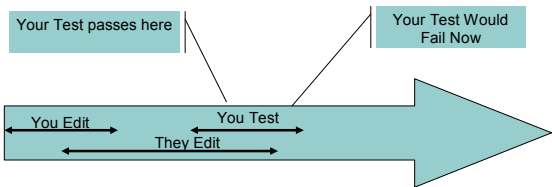
---

---

---

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*.

---

---

---

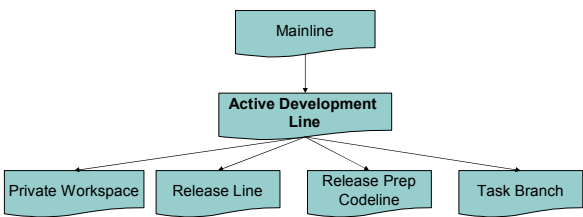
---

---

---

---

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*.

---

---

---

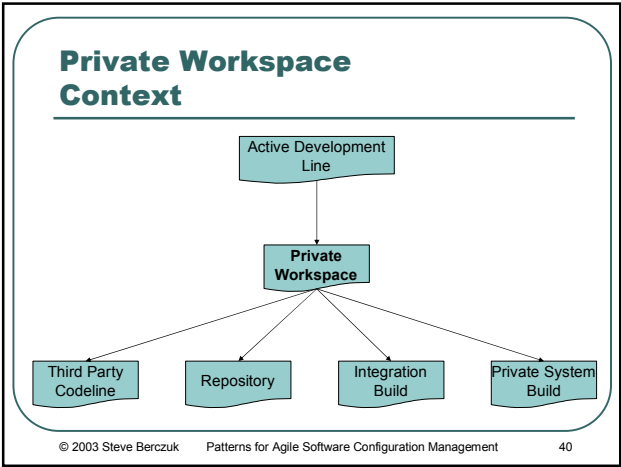
---

---

---

---

# Patterns for Agile Software Configuration Management



---

---

---

---

---


---

---

---

Repository

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



© 2003 Steve Berczuk

Patterns for Agile Software Configuration Management

41

---

---

---

---

---

---

---

---

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 (3<sup>rd</sup> Parties, other groups, etc).

© 2003 Steve Berczuk

Patterns for Agile Software Configuration Management

42

---

---

---

---

---

---

---

---

Repository (Solution)

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

---

---

---

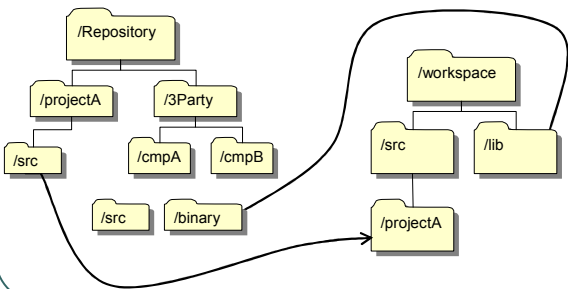
---

---

---

---

Mapping from Repository to Workspace



---

---

---

---

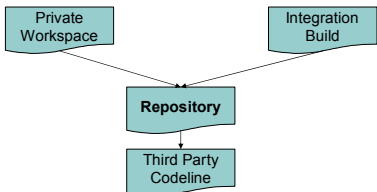
---

---

---

Repository (Unresolved)

- Manage external components: *Third Party Codeline*



---

---

---

---

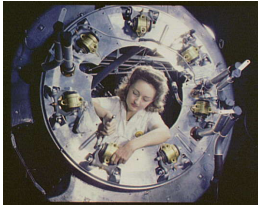
---

---

---

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 needs than the system build.
- 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.

---

---

---

---

---

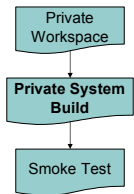
---

---

---

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 Builds* are a way to check the build.
- Building everything may take a long time.
- You want to ensure that what is checked-in works.

---

---

---

---

---

---

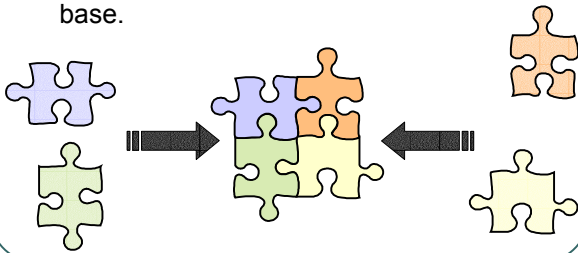
---

---

# Patterns for Agile Software Configuration Management

## Integration Build (Solution)

- Do a centralized build for the entire code base.



© 2003 Steve Berczuk Patterns for Agile Software Configuration Management 52

---

---

---

---

---

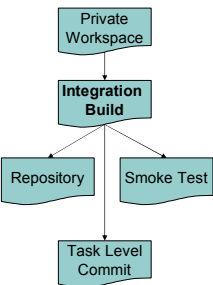
---

---

---

## 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*.



© 2003 Steve Berczuk Patterns for Agile Software Configuration Management 53

---

---

---

---

---

---

---

---

## Third Party Codeline

- Private Workspaces* and the *Repository* need the right versions of external components.
- How do you coordinate versions of external components with your versions?



© 2003 Steve Berczuk Patterns for Agile Software Configuration Management 54

---

---

---

---

---

---

---

---

Third Party Codeline  
(Forces & Tradeoffs)

- Vendor releases do not match your releases.
- Sometimes you alter external code (open source, etc) or apply patches.

---

---

---

---

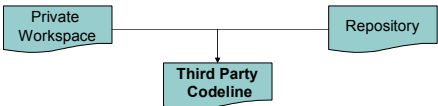
---

---

---

Third Party Codeline (Solution)

- Use the same mechanisms as you do for your code to create a *Third Party Codeline*.
- Label the codeline to associate snapshots with your versions.



---

---

---

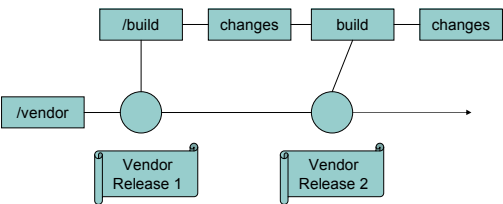
---

---

---

---

Third Party Codeline (Structure)



---

---

---

---

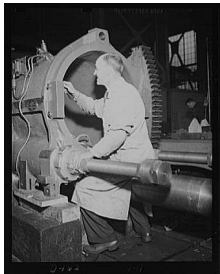
---

---

---

**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 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?

---

---

---

---

---

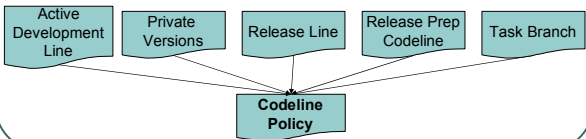
---

---

---

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.



---

---

---

---

---

---

---

---

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
  - 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.

---

---

---

---

---

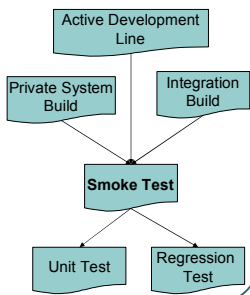
---

---

---

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*



---

---

---

---

---

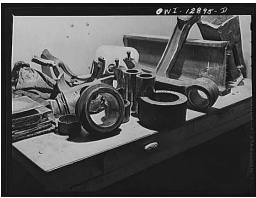
---

---

---

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.

---

---

---

---

---

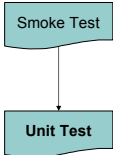
---

---

---

Unit Test (Solution)

- Develop and run *Unit Tests*



---

---

---

---

---

---

---

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.

---

---

---

---

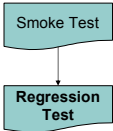
---

---

---

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.



---

---

---

---

---

---

---

Private Versions

- An *Active Development Line* will break if people check in half-finished tasks.
- How can you experiment with complex changes and still get the benefits of version management?



---

---

---

---

---

---

---

Private Versions  
(Forces & Tradeoffs)

- Sometimes you may want to checkpoint an intermediate step of a long, complex change.
- Your version management system provides the facilities for checkpointing.
- You don't want to publish intermediate steps.

---

---

---

---

---

---

---

Private Versions (Solution)

- Provide developers with a mechanism for checkpointing changes using a simple interface.
- Implement as:
  - Private History
  - A Private Repository
  - A Private Branch
- [Compare with Task Branch for long lived /joint efforts.]

---

---

---

---

---

---

---

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.

---

---

---

---

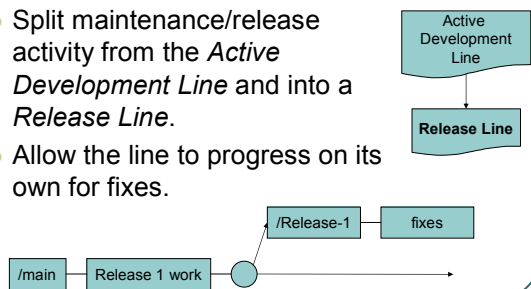
---

---

---

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.



---

---

---

---

---

---

---

---

Release Prep Codeline

- You want to maintain an *Active Development Line*.
- How do you stabilize a codeline for an imminent release while allowing new work to continue on an active codeline?



---

---

---

---

---

---

---

---

Release-Prep Codeline (Forces & Tradeoffs)

- You want to stabilize a codeline so you can ship it.
- A code freeze slows things down too much.
- Branches have overhead.

---

---

---

---

---

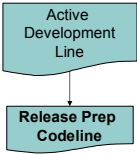
---

---

---

Release Prep Codeline  
(Solution)

- Branch instead of freeze. Create a *Release Prep Codeline* (a branch) when code is approaching release quality.
- Leave the *Mainline* for active development.
- The *Release Prep Codeline* becomes the *Release Line* (with a stricter policy)
- Note: If only a few people are doing work on the next release, consider a *Task Branch* instead.



---

---

---

---

---

---

---

Task Branch

- Some tasks have intermediate steps that would disrupt an *Active Development Line*.
- **How can your team make multiple, long-term, overlapping changes to a codeline without compromising its integrity?**



---

---

---

---

---

---

---

Task Branch  
(Forces & Tradeoffs)

- Version Management is a communication mechanism.
- Sometimes only part of a team is working on a task.
- Some changes have many steps.
- Branching has overhead.

---

---

---

---

---

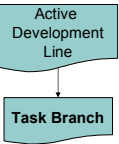
---

---

# Patterns for Agile Software Configuration Management

## Task Branch (Solution)

- Create a *Task Branch* off of the *Mainline* for each activity that has significant changes for a codeline.
- Integrate this codeline back into the *Mainline* when done.
- Be sure to integrate changes from the *Mainline* into this codeline as you go.
- [*Compare with Private Versions.*]



---

---

---

---

---

---

---

## Moving Forward: Wrap Up



---

---

---

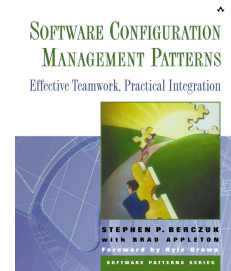
---

---

---

---

## Our Book



- Pub Nov 2002 By Addison-Wesley Professional.

---

---

---

---

---

---

---

Other Pointers

- [www.scmpatterns.com](http://www.scmpatterns.com)
- [acme.bradapp.net](http://acme.bradapp.net)
- [www.berczuk.com](http://www.berczuk.com)
- [www.cmcrossroads.com](http://www.cmcrossroads.com)
- [steve@berczuk.com](mailto:steve@berczuk.com)



---

---

---

---

---

---

---