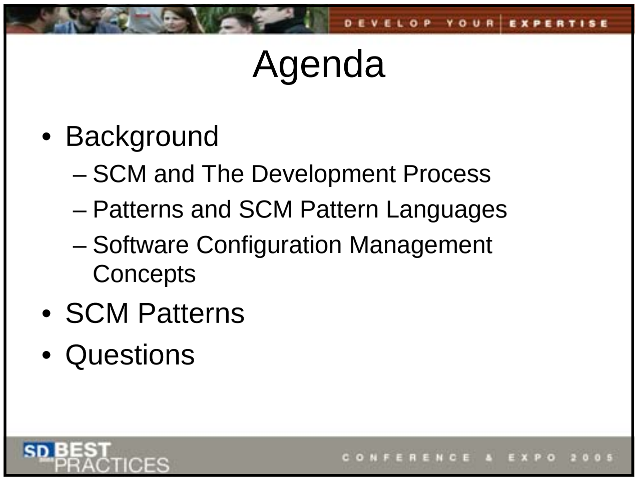
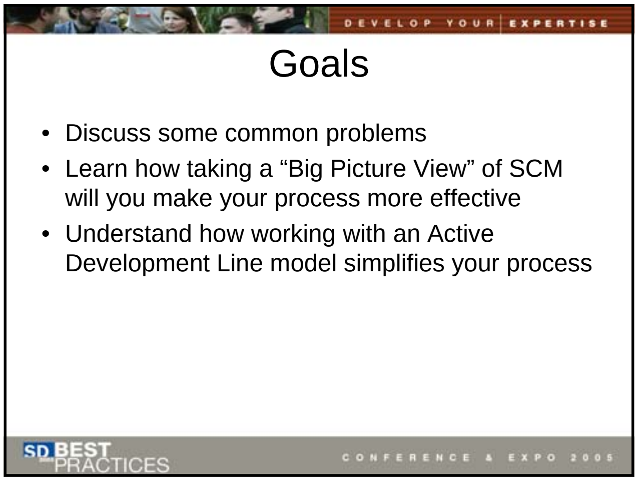
Collaborate, Build, Test, Deploy: Essential SCM Practices for Teams

Steve Berczuk
September 28, 2005
SD Best Practices



Agenda

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



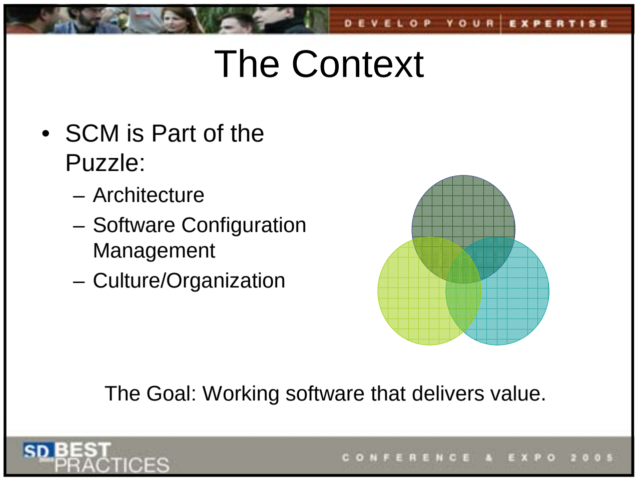
Goals

- 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

About Me

- Software Developer, Architect, Consultant, Author. Currently at Iron Mountain Digital
- Startup and established company experience
- Systems ranging from travel web sites, to enterprise systems, to space science systems
- Agile and Iterative Development

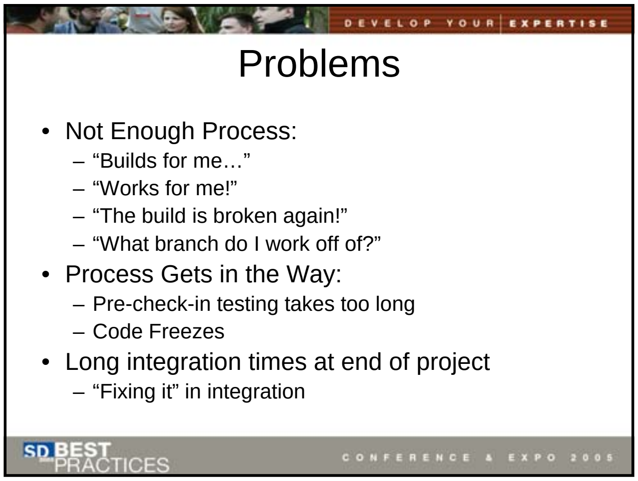
Foundations



The Context

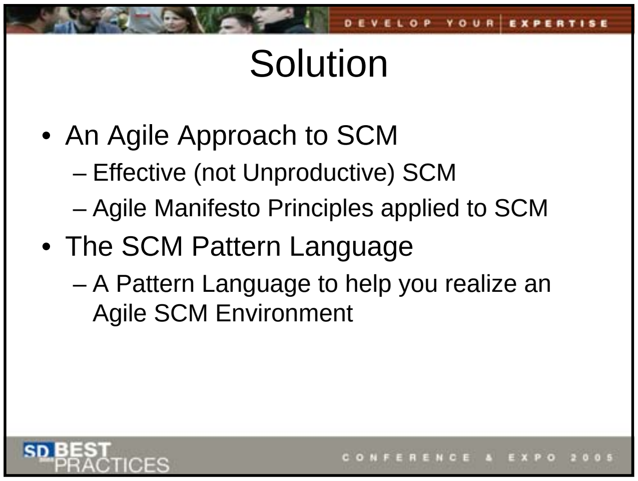
- SCM is Part of the Puzzle:
  - Architecture
  - Software Configuration Management
  - Culture/Organization

The Goal: Working software that delivers value.


DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

Problems

- Not Enough Process:
  - “Builds for me…”
  - “Works for me!”
  - “The build is broken again!”
  - “What branch do I work off of?”
- Process Gets in the Way:
  - Pre-check-in testing takes too long
  - Code Freezes
- Long integration times at end of project
  - “Fixing it” in integration


DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

Solution

- An Agile Approach to SCM
  - Effective (not Unproductive) SCM
  - Agile Manifesto Principles applied to SCM
- The SCM Pattern Language
  - A Pattern Language to help you realize an Agile SCM Environment

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.

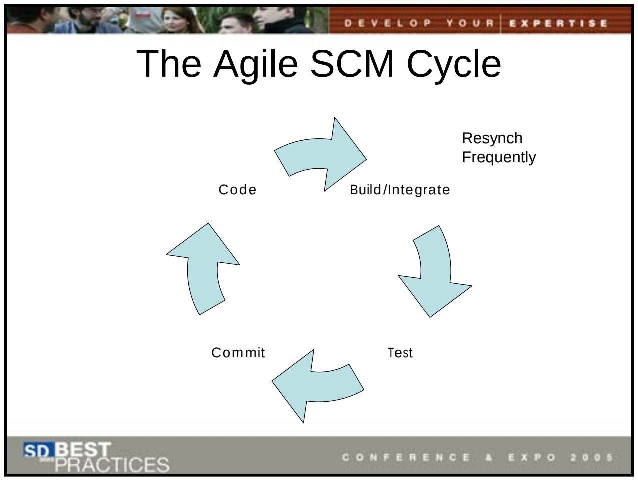
What is *Agile SCM?*

- *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
- *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

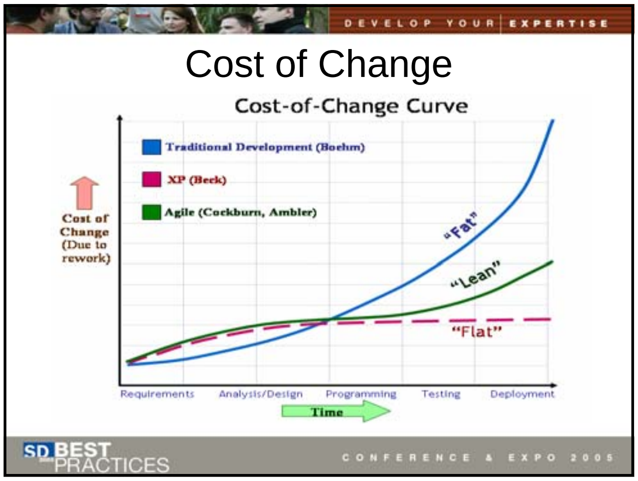
What Agile SCM is Not

- Lack of process
- Chaos
- Lack of control

Agile SCM is about having an Effective SCM process that helps get work done.

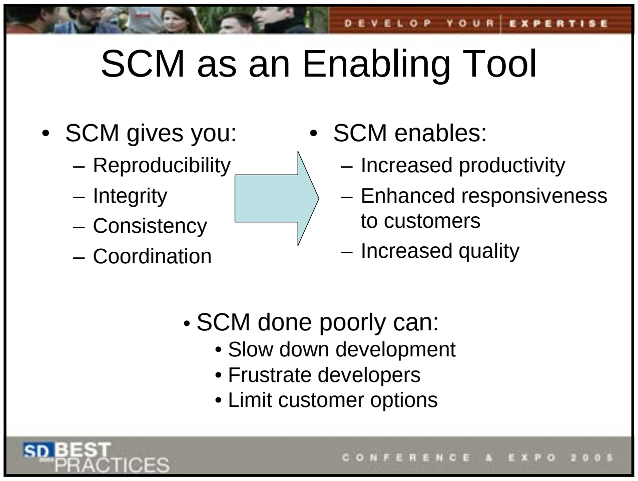


The Agile SCM Cycle



Cost of Change

SCM Concepts



SCM as an Enabling Tool

- SCM gives you:
  - Reproducibility
  - Integrity
  - Consistency
  - Coordination
- SCM enables:
  - Increased productivity
  - Enhanced responsiveness to customers
  - Increased quality

- SCM done poorly can:
  - Slow down development
  - Frustrate developers
  - Limit customer options

DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

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

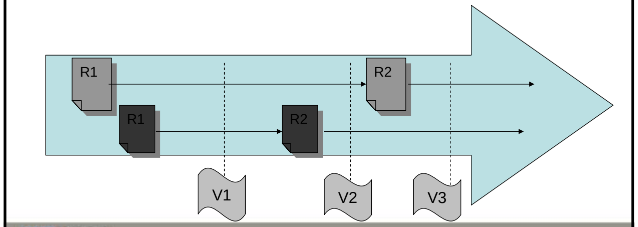
- Frequent feedback on build quality and product suitability through:
  - Version Management
  - Release Management
  - Build Management
  - Unit & Regression Testing

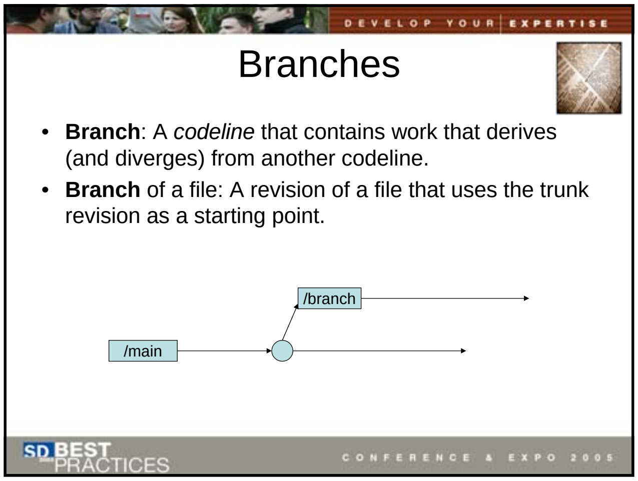
SCM Definitions

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

Codeline

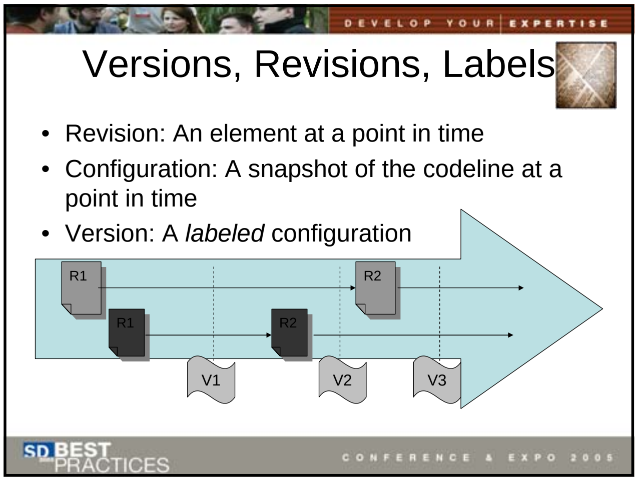
- A **codeline** contains every version of every artifact over one evolutionary path.





Branches

- **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.

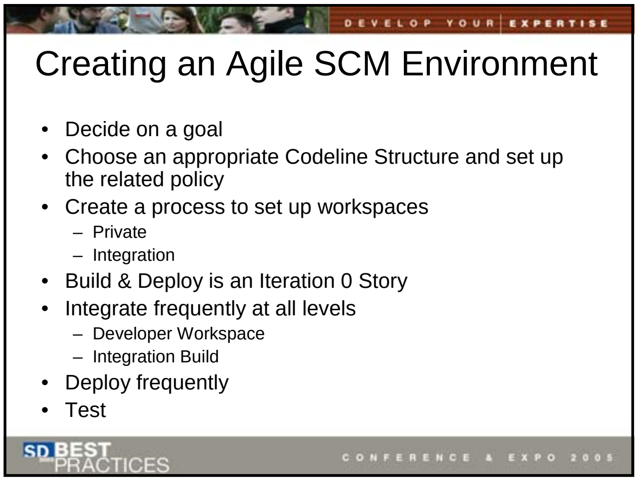


Versions, Revisions, Labels

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

Definition: Workspace

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



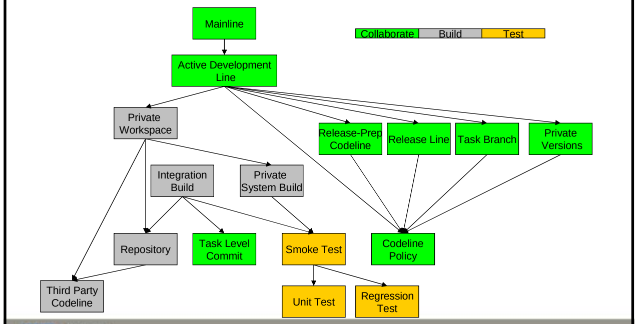
Creating an Agile SCM Environment

- Decide on a goal
- Choose an appropriate Codeline Structure and set up 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

Collaborate/Build/Test

- Collaboration Patterns
  - Workspaces
  - Codelines
  - Unit of Work
- Build Patterns
- Test Patterns

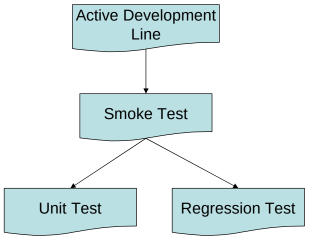
The SCM Pattern Language



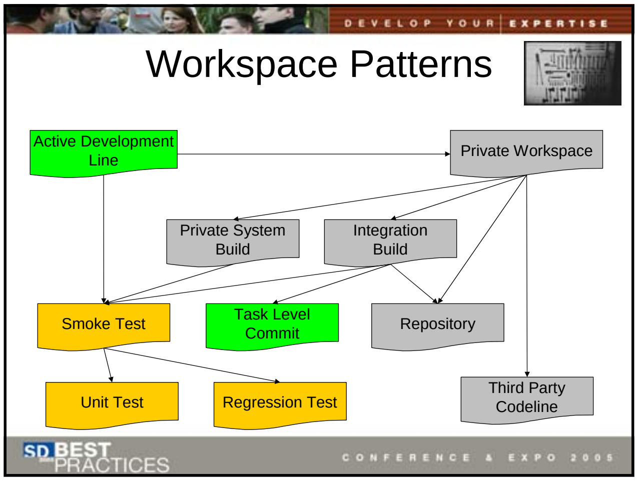
What are *Patterns* and *Pattern Languages*?

- 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

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



Workspace Patterns

Active Development Line

Private Workspace

Private System Build

Integration Build

Smoke Test

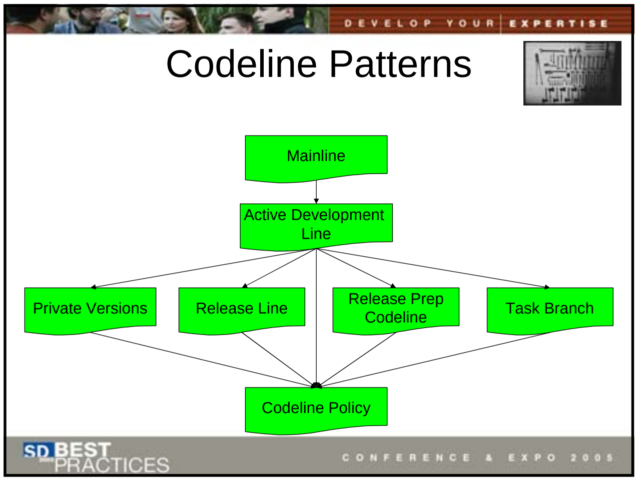
Task Level Commit

Repository

Unit Test

Regression Test

Third Party Codeline



Codeline Patterns

Mainline

Active Development Line

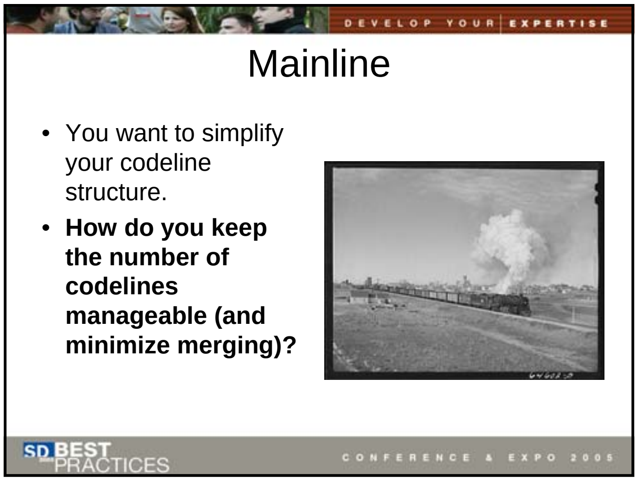
Private Versions

Release Line

Release Prep Codeline

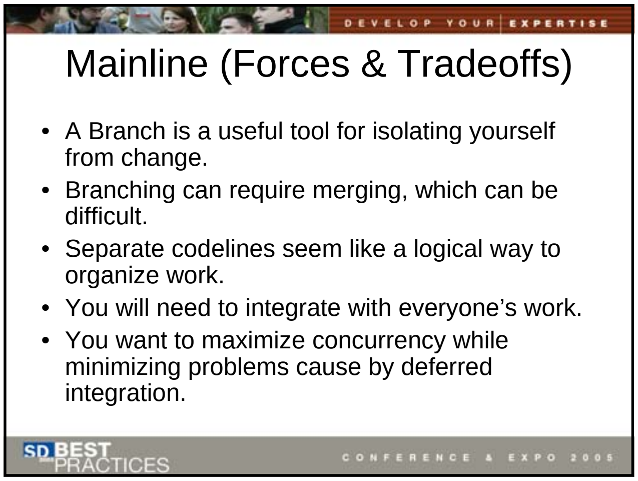
Task Branch

Codeline Policy



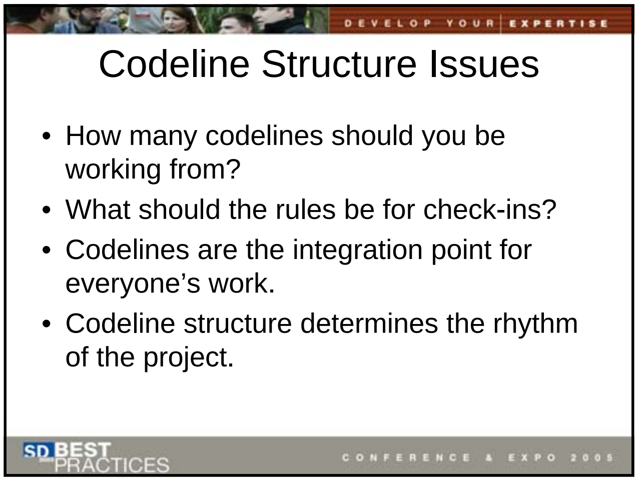
Mainline

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



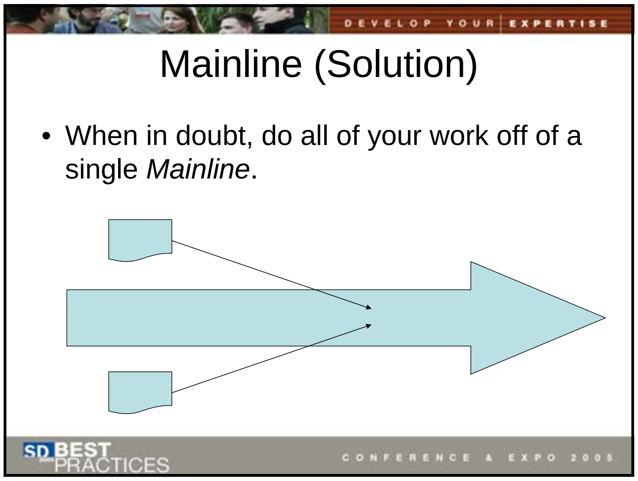
Mainline (Forces & Tradeoffs)

- 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 with everyone’s work.
- You want to maximize concurrency while minimizing problems cause by deferred integration.



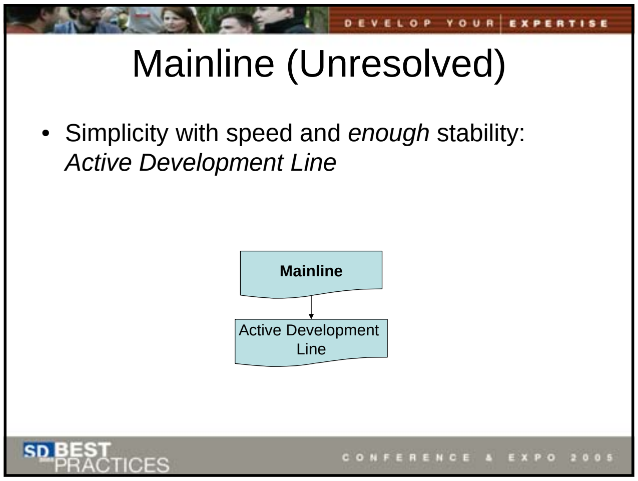
Codeline Structure Issues

- 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 (Solution)

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

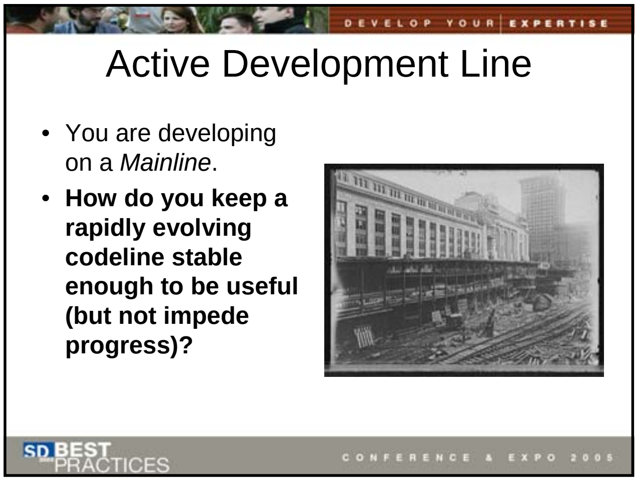


Mainline (Unresolved)

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

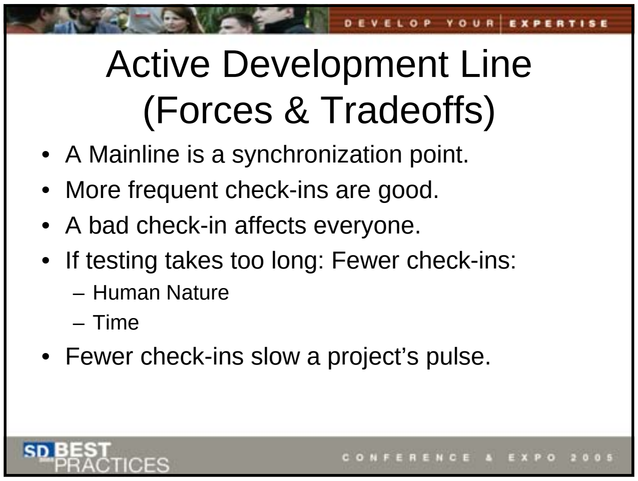
Mainline

Active Development Line



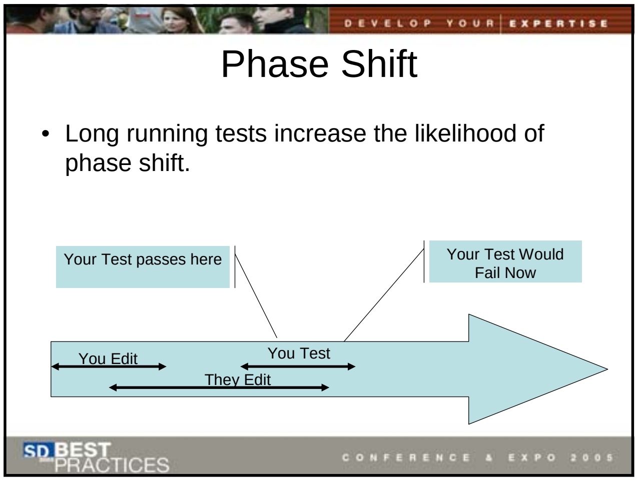
Active Development Line

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


DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

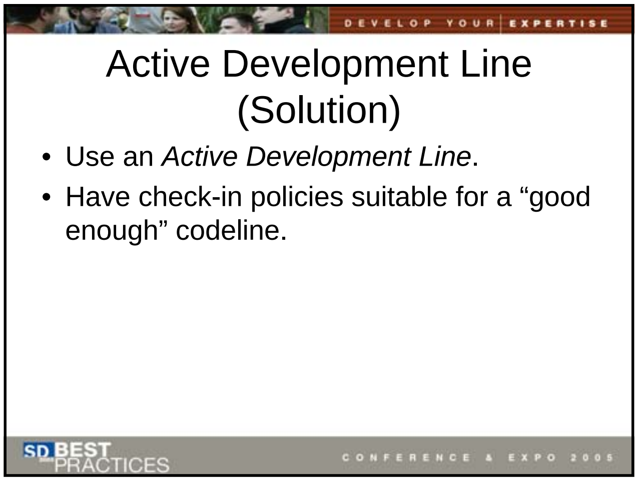
Active Development Line (Forces & Tradeoffs)

- 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 a project's pulse.



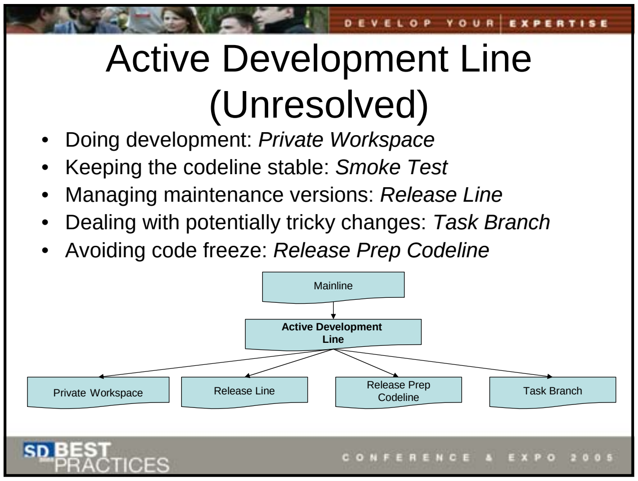
Phase Shift

- Long running tests increase the likelihood of phase shift.


DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

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*

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?**

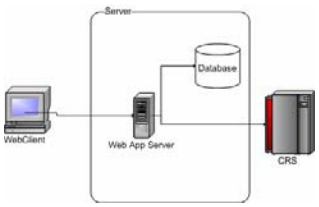
Private Workspace (Forces & Tradeoffs)

- 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

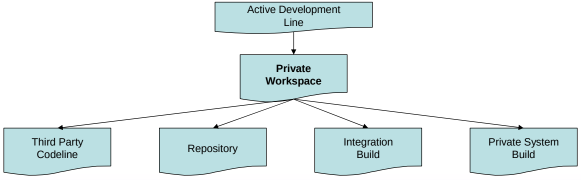
Private Workspace Example



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

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*



Repository

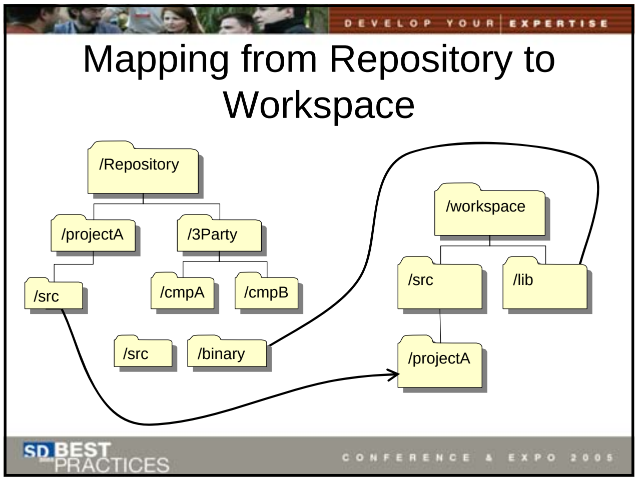
- *Private Workspace* and *Integration Build* need components.
- **How do you get the right versions of the right components into a new workspace?**

Repository (Forces & Tradeoffs)

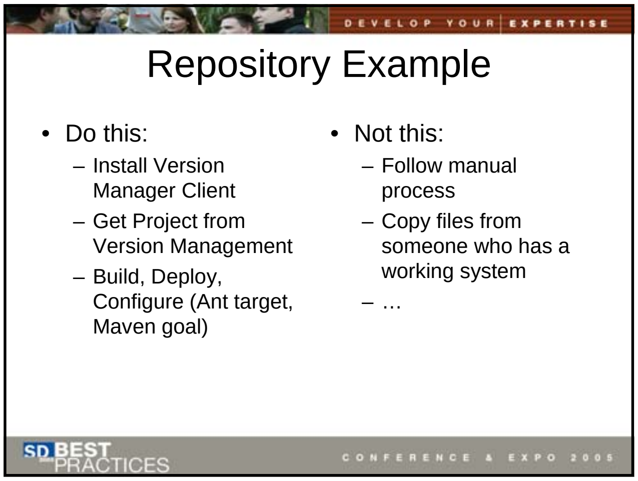
- Many things make up a workspace: code, libraries, scripts.
- You want to be able to easily build a workspace from nothing.
- These components could come from a variety of sources (3rd Parties, other groups, etc).

Repository (Solution)

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

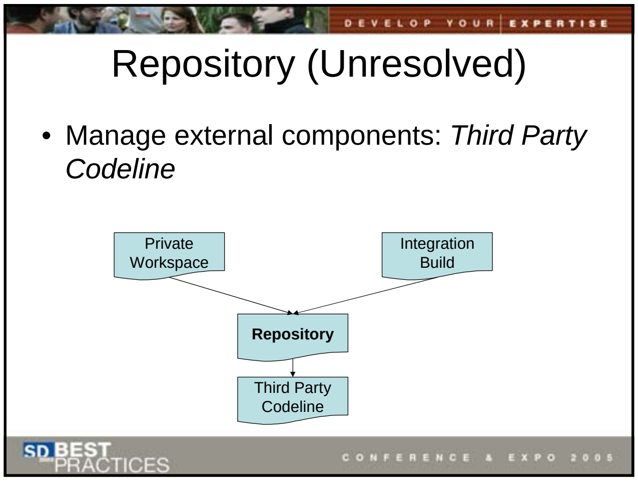


Mapping from Repository to Workspace



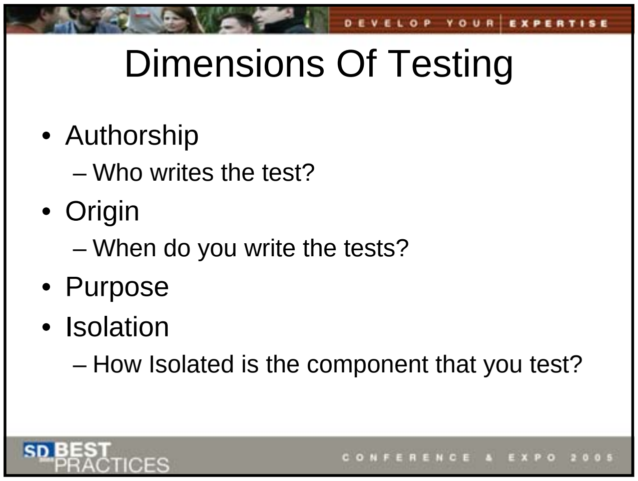
Repository Example

- Do this:
  - Install Version Manager Client
  - Get Project from Version Management
  - Build, Deploy, Configure (Ant target, Maven goal)
- Not this:
  - Follow manual process
  - Copy files from someone who has a working system
  - …



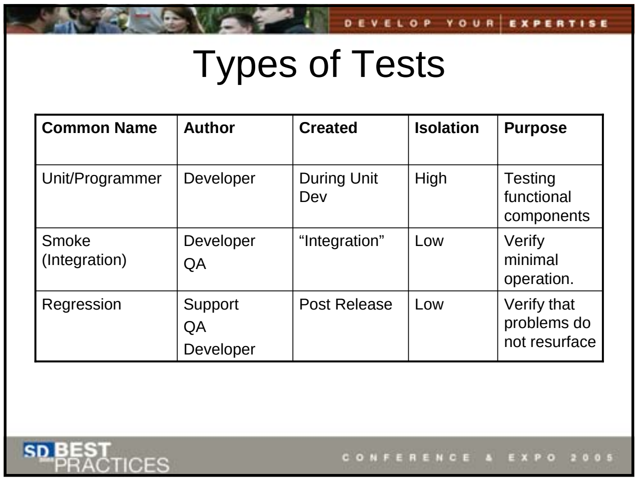
Repository (Unresolved)

- Manage external components: *Third Party Codeline*



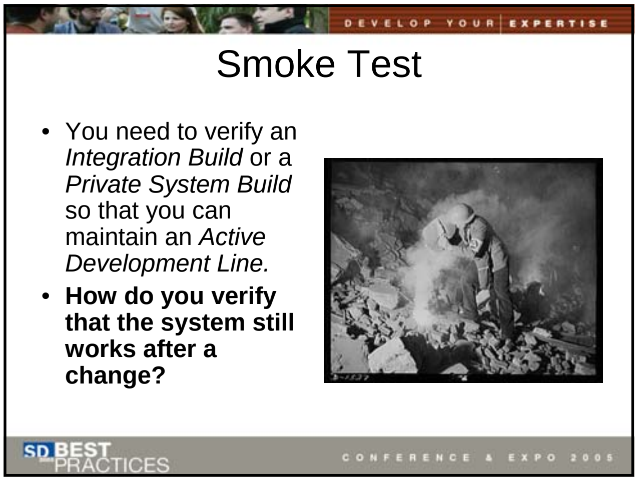
Dimensions Of Testing

- Authorship
  - Who writes the test?
- Origin
  - When do you write the tests?
- Purpose
- Isolation
  - How Isolated is the component that you test?



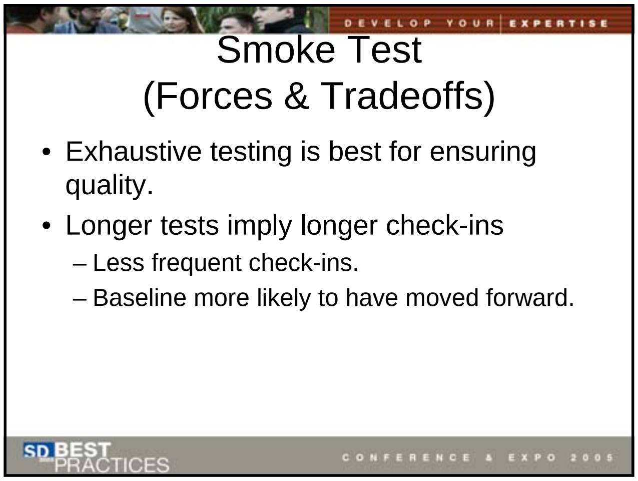
Types of Tests

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



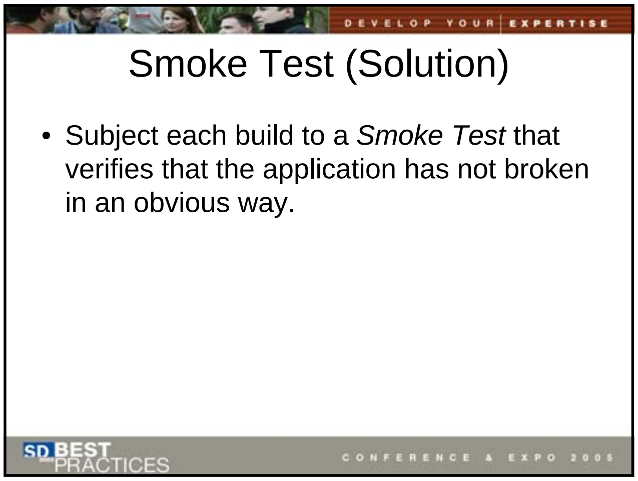
Smoke Test

- 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?**



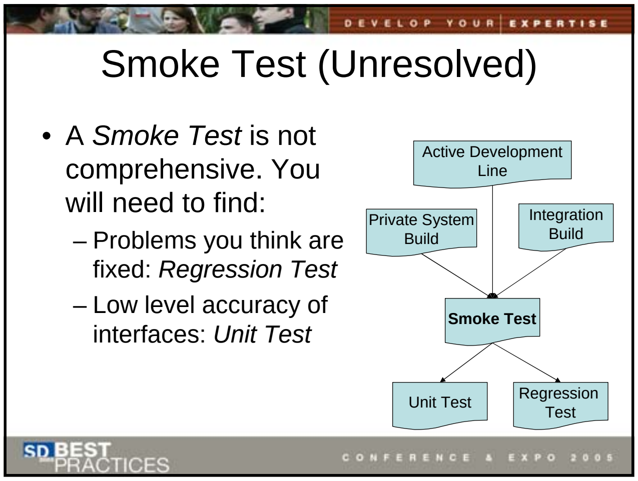
Smoke Test (Forces & Tradeoffs)

- Exhaustive testing is best for ensuring quality.
- Longer tests imply longer check-ins
  - 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*

Active Development Line

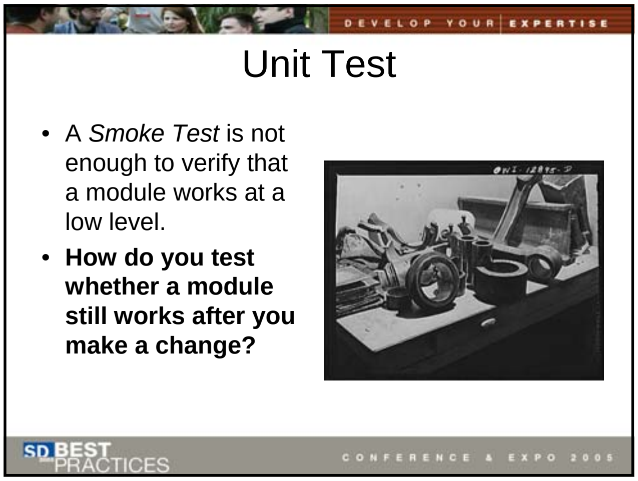
Private System Build

Integration Build

Smoke Test

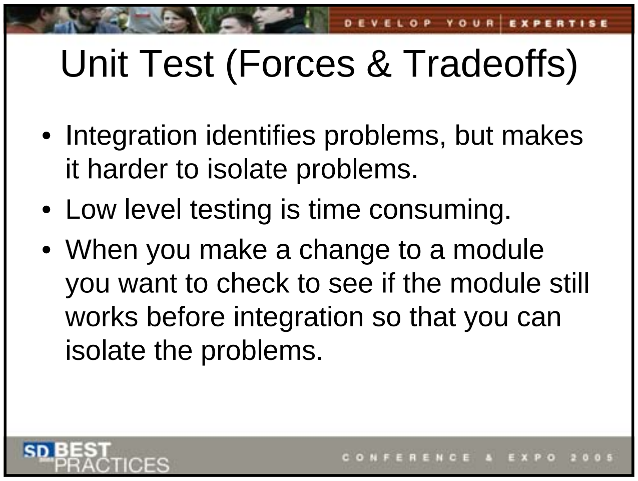
Unit Test

Regression Test



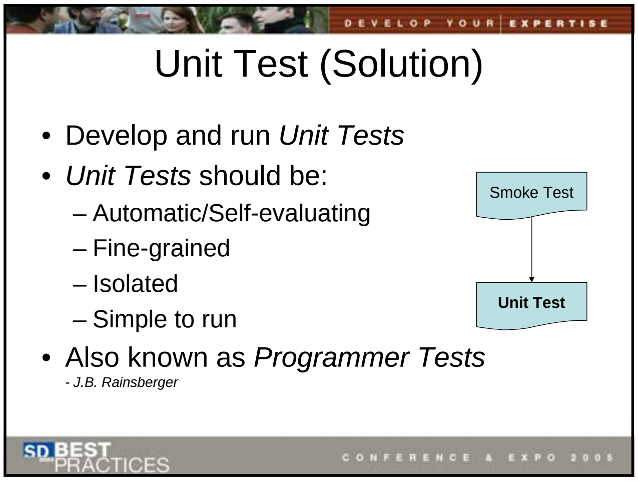
Unit Test

- 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.



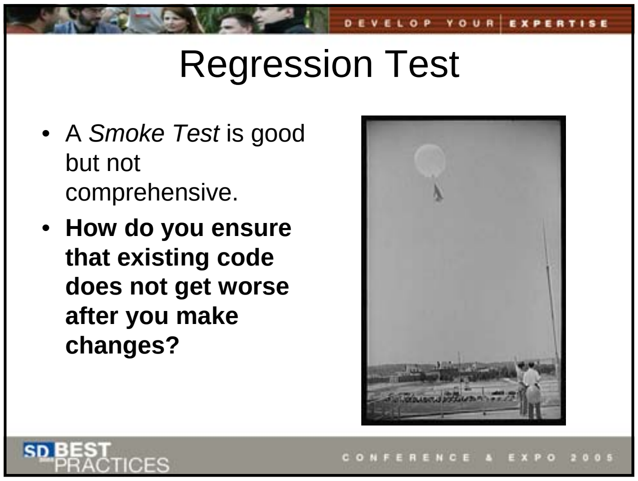
Unit Test (Solution)

- 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

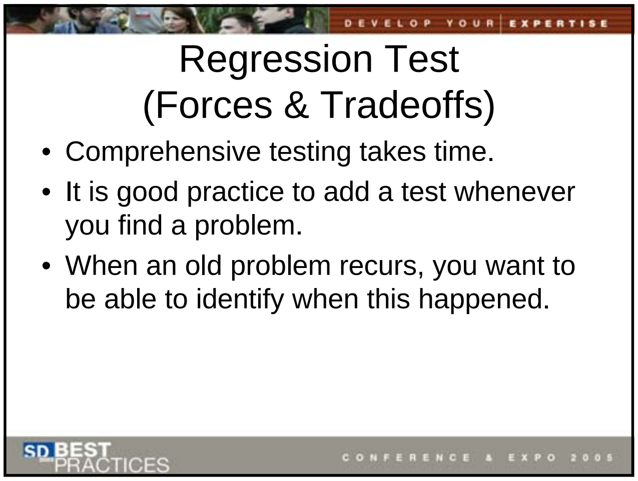
Smoke Test

Unit Test


DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

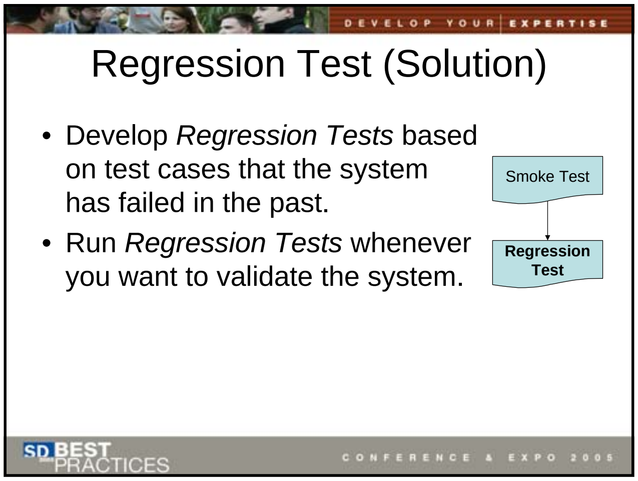
Regression Test

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


DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

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.

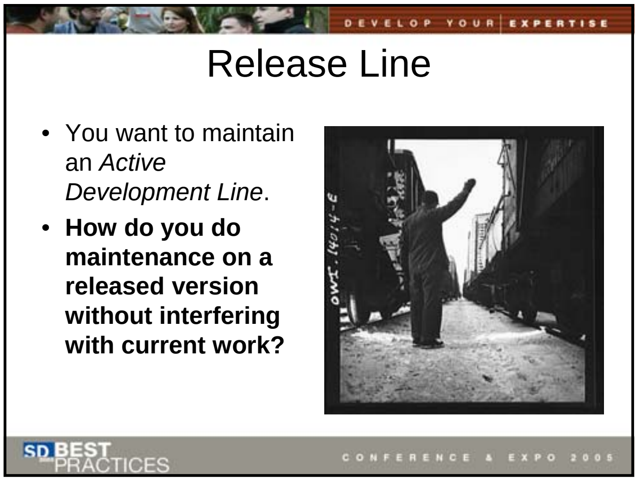

DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

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.

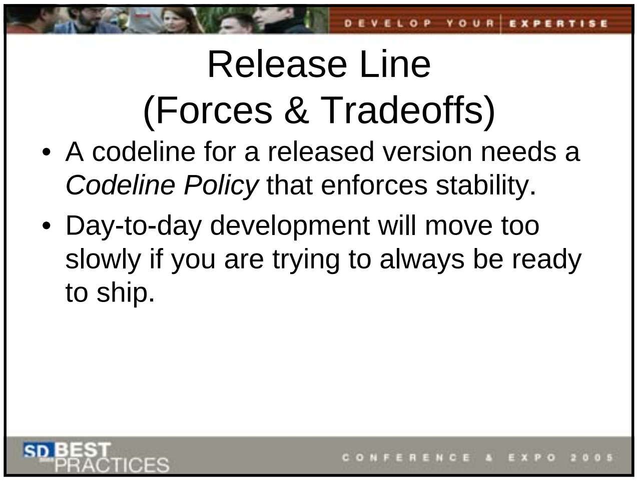
Smoke Test

Regression Test


DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

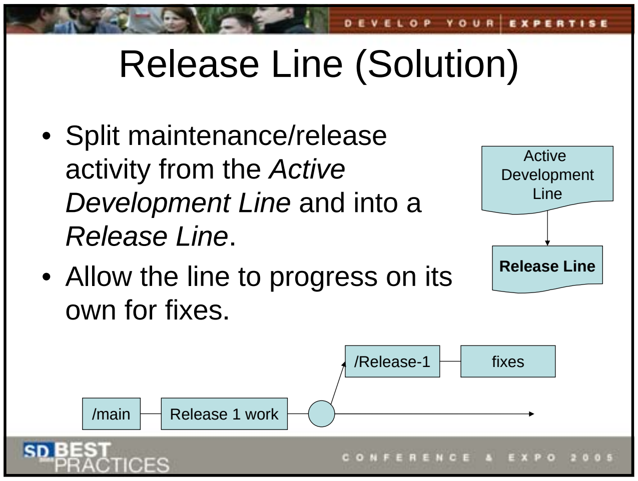
Release Line

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


DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

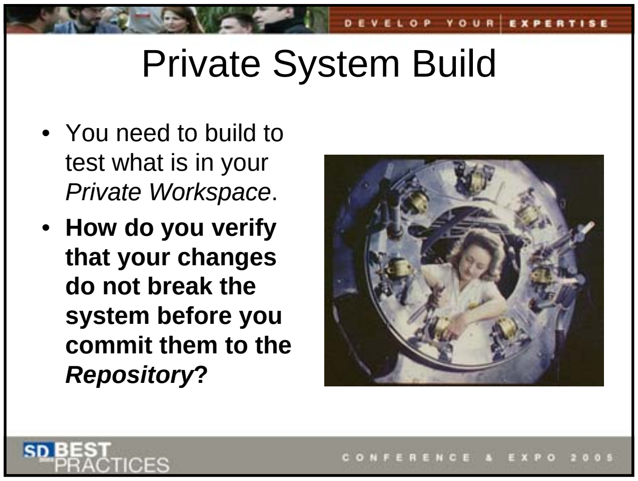
Release Line (Forces & Tradeoffs)

- 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.



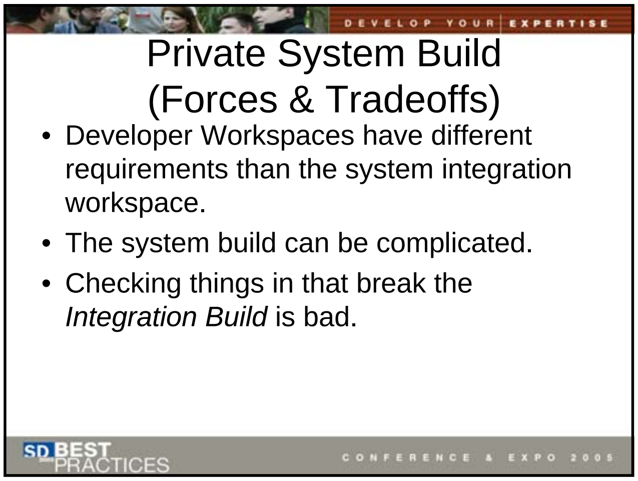
Release Line (Solution)

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


DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

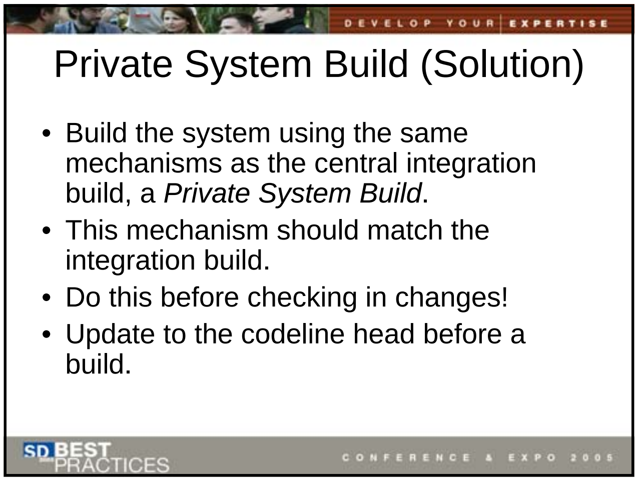
Private System Build

- 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*?**


DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

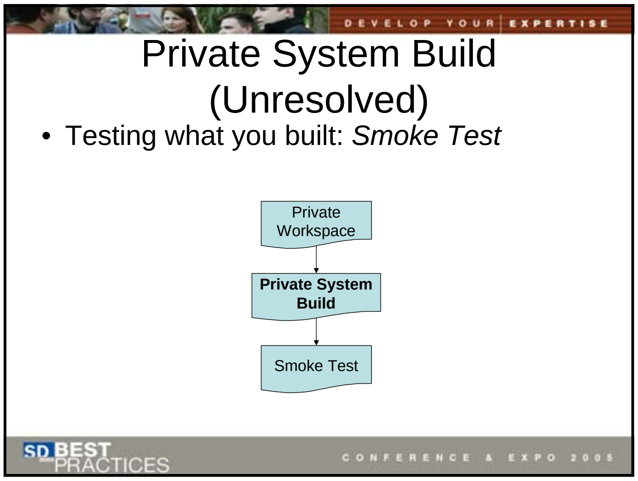
Private System Build (Forces & Tradeoffs)

- 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.


DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

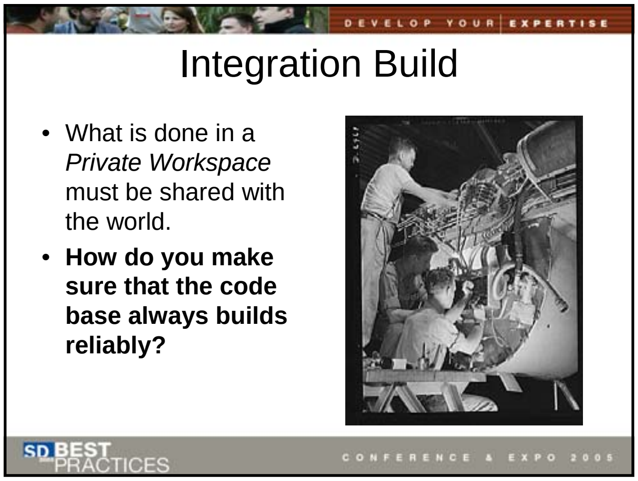
Private System Build (Solution)

- 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.



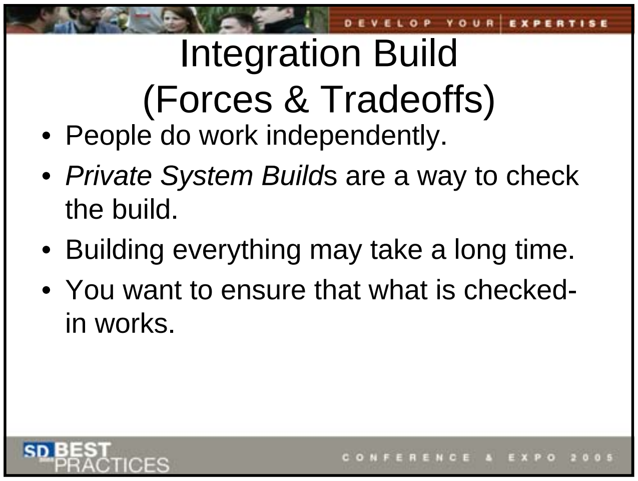
Private System Build (Unresolved)

- Testing what you built: *Smoke Test*


DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

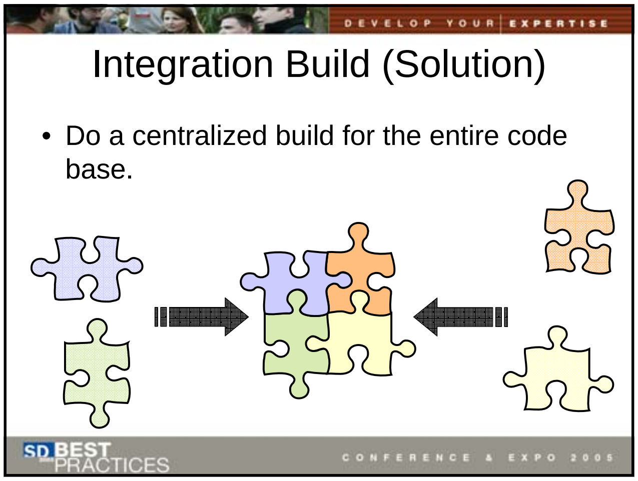
Integration Build

- 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?**


DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

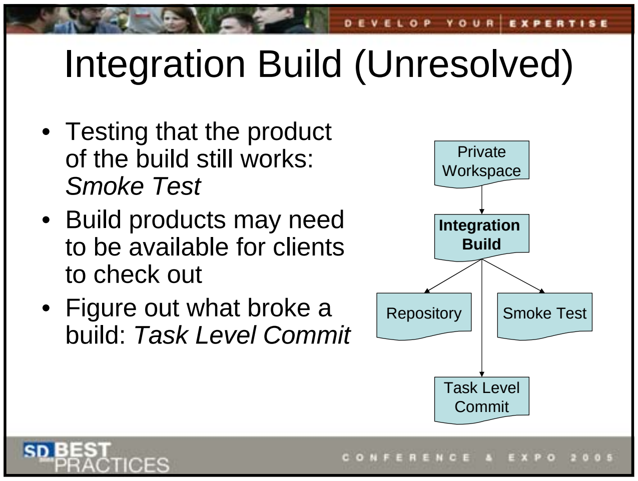
Integration Build (Forces & Tradeoffs)

- 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.


DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

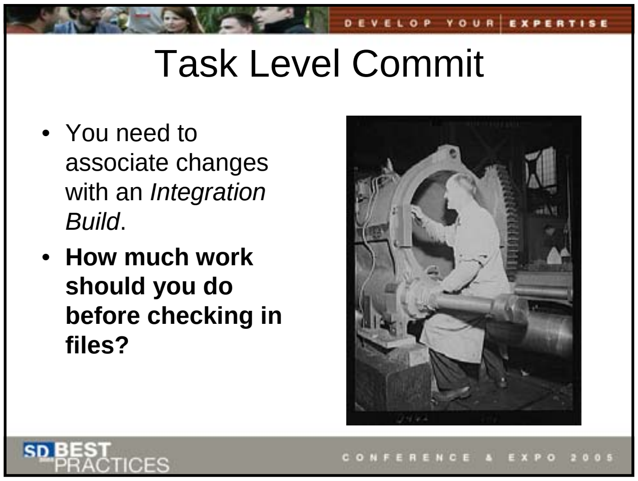
Integration Build (Solution)

- Do a centralized build for the entire code base.


DEVELOP YOUR EXPERTISE
Private Workspace
Integration Build
Repository
Smoke Test
Task Level Commit
SD BEST PRACTICES
CONFERENCE & EXPO 2005

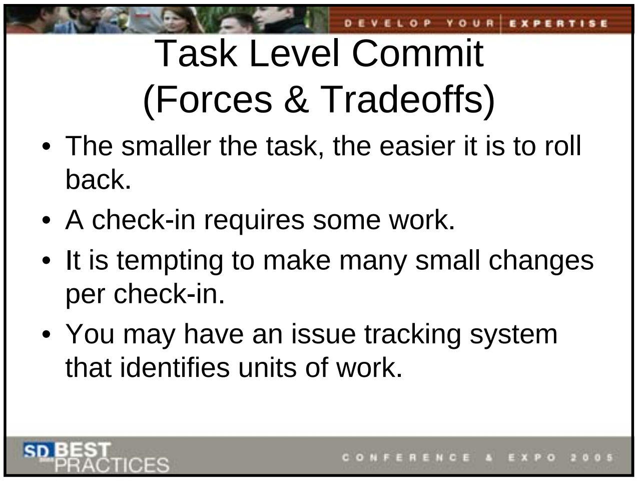
Integration Build (Unresolved)

- 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*


DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

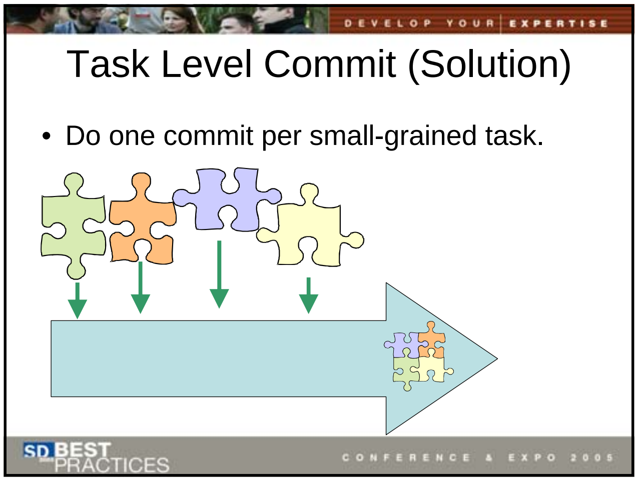
Task Level Commit

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


DEVELOP YOUR EXPERTISE
SD BEST PRACTICES
CONFERENCE & EXPO 2005

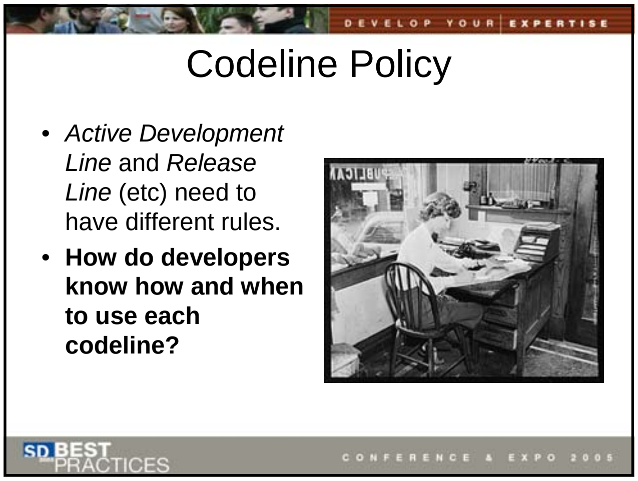
Task Level Commit (Forces & Tradeoffs)

- 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.



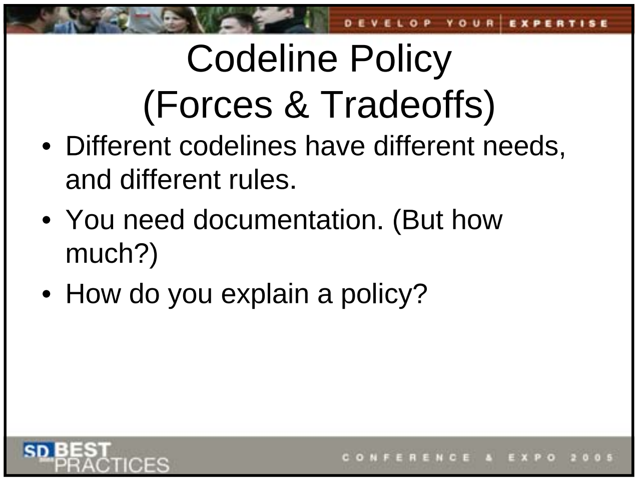
Task Level Commit (Solution)

- Do one commit per small-grained task.



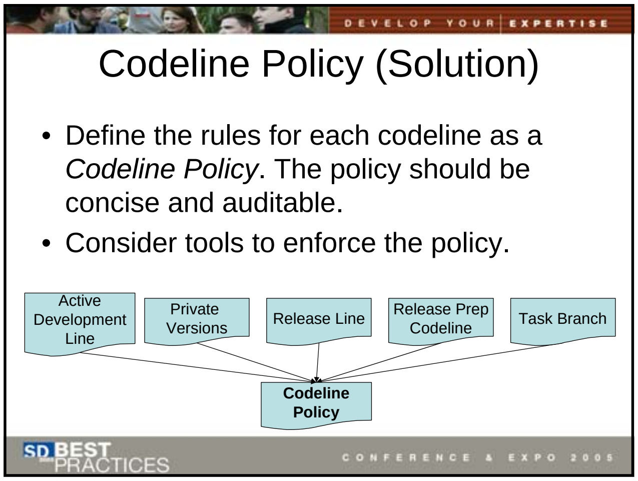
Codeline Policy

- *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.

Active Development Line

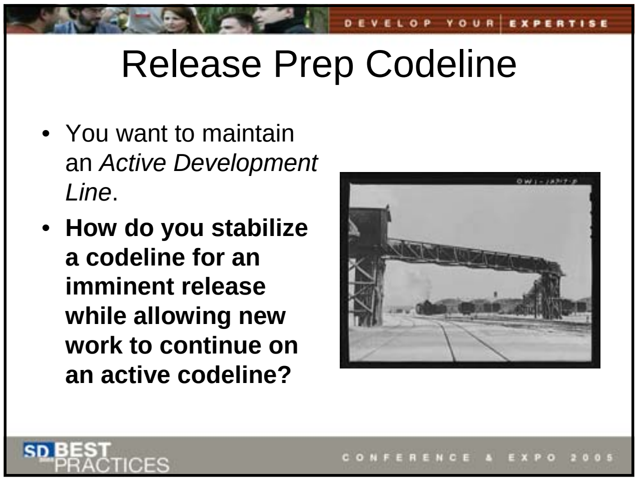
Private Versions

Release Line

Release Prep Codeline

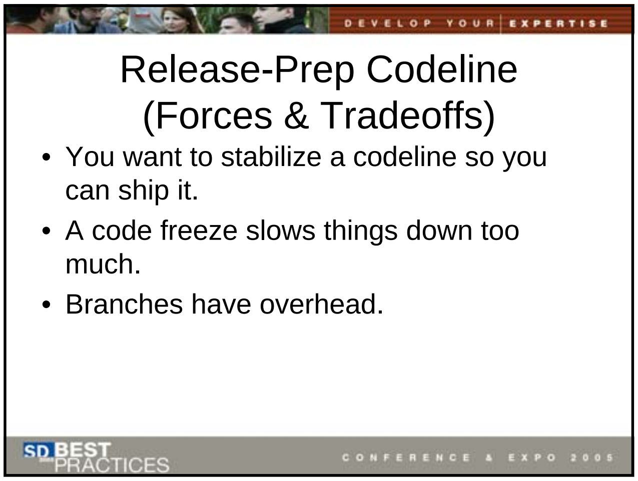
Task Branch

Codeline Policy



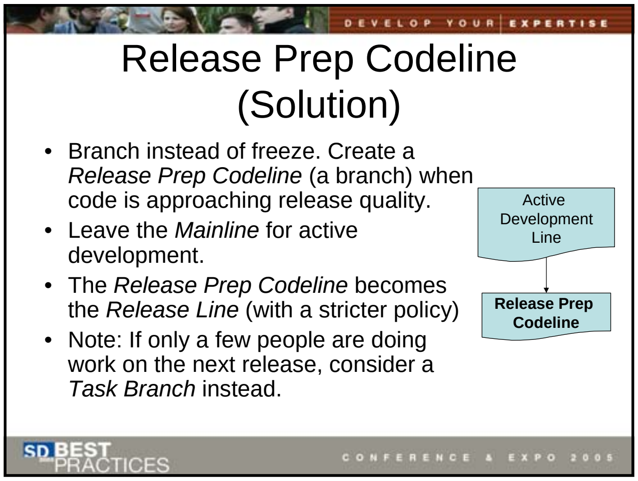
Release Prep Codeline

- 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.

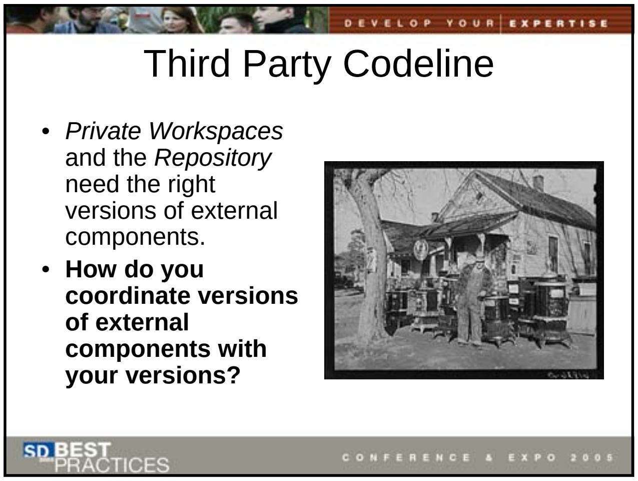


Release Prep Codeline (Solution)

- 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.

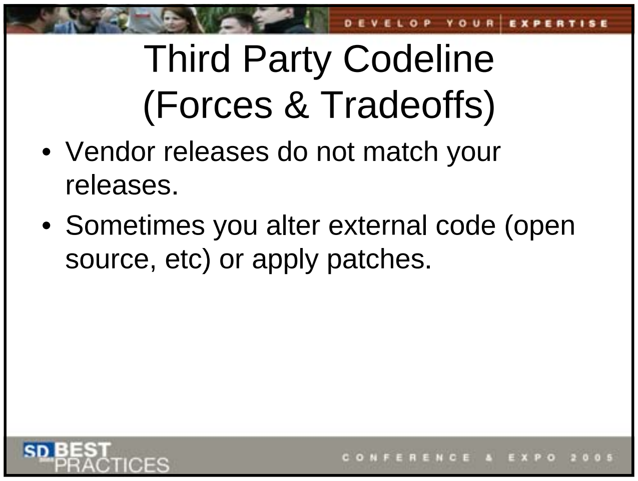
Active Development Line

Release Prep Codeline



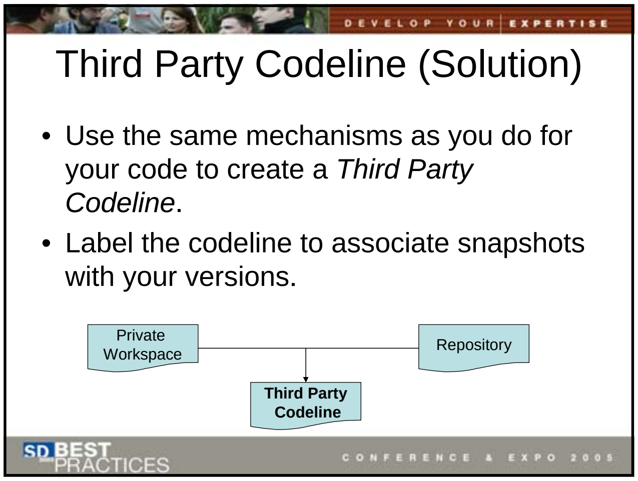
Third Party Codeline

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



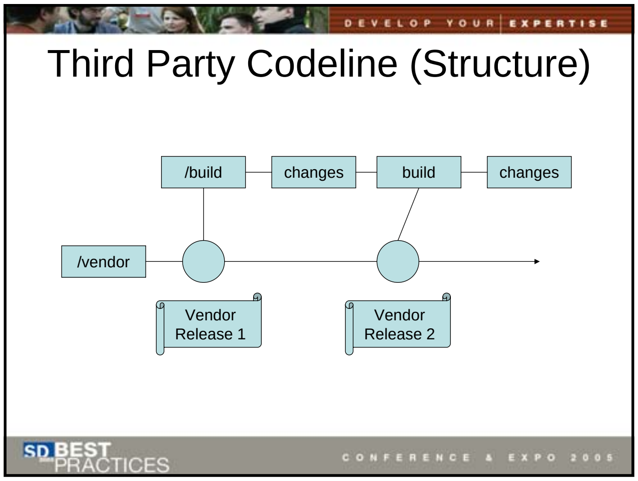
Third Party Codeline (Forces & Tradeoffs)

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

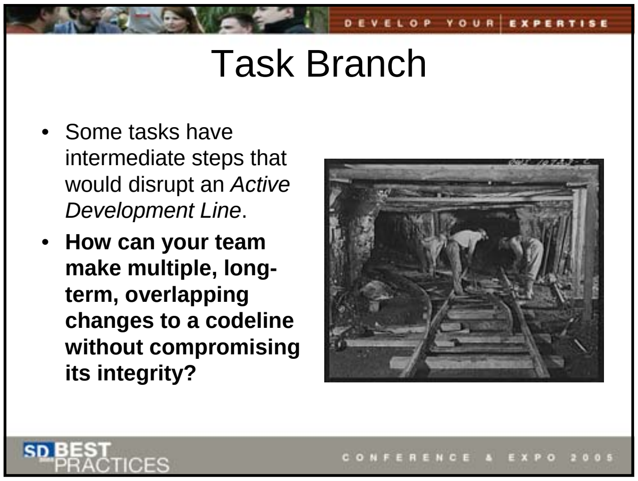


Third Party Codeline (Solution)

- 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.

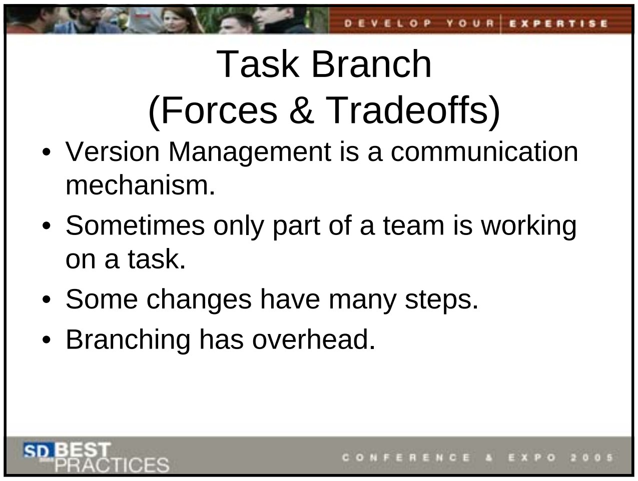


Third Party Codeline (Structure)



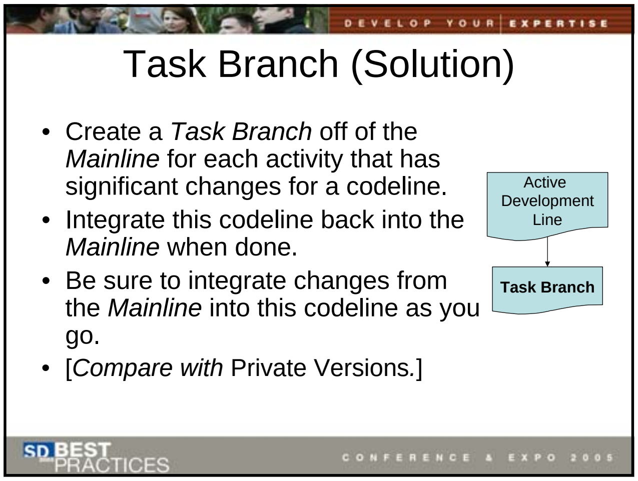
Task Branch

- 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.

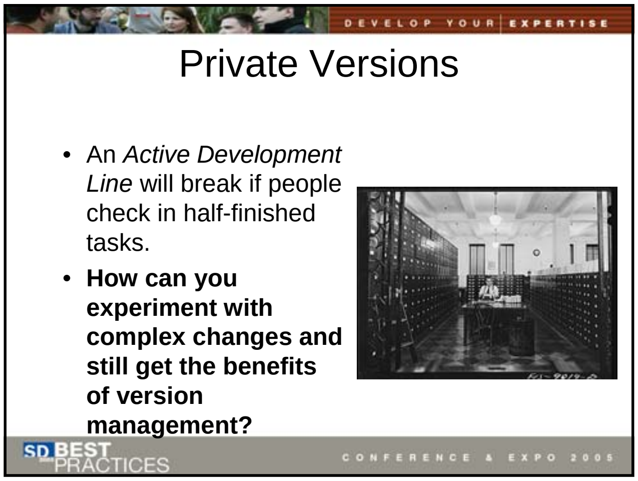


Task Branch (Solution)

- 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.]

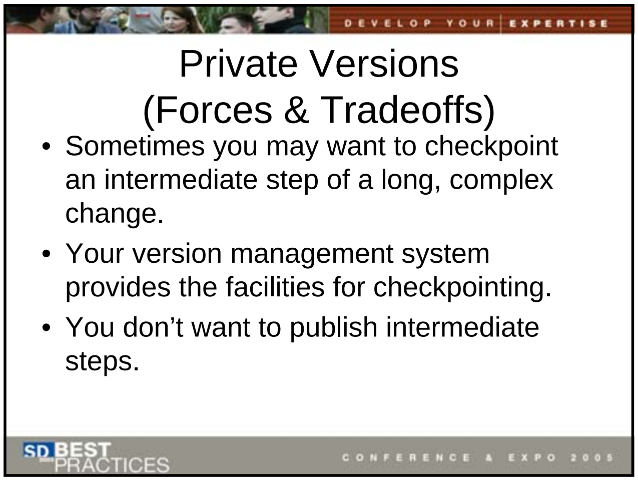
Active Development Line

Task Branch



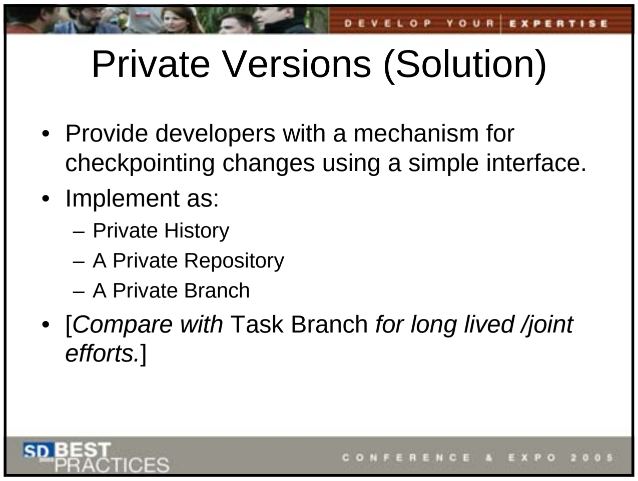
Private Versions

- 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)

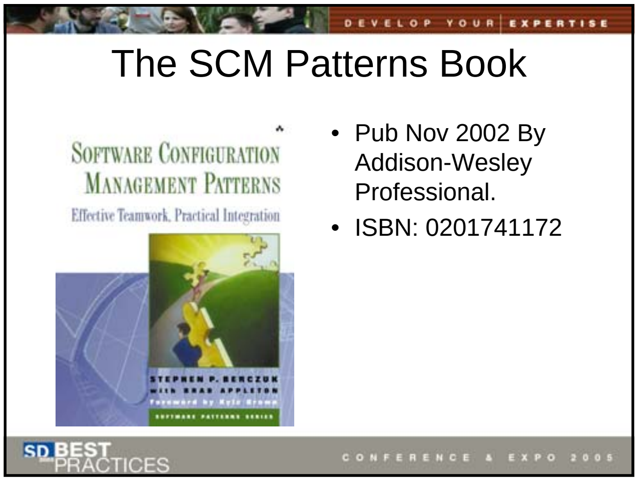
- Sometimes you may want to checkpoint an intermediate step of a long, complex change.
- Your version management system provides the facilities for checkpointing.
- You don’t want to publish intermediate steps.



Private Versions (Solution)

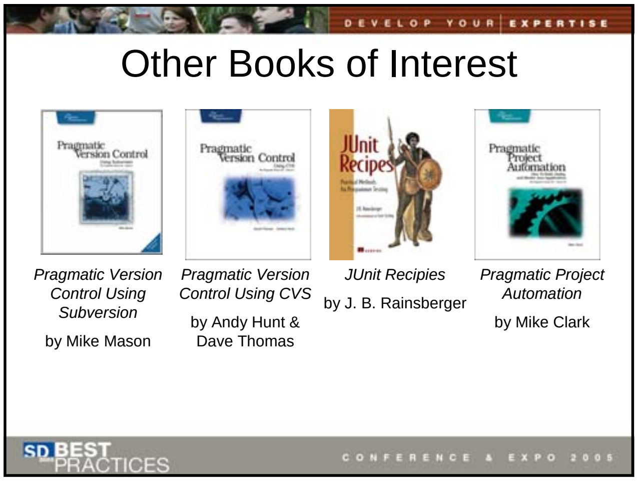
- Provide developers with a mechanism for checkpointing changes using a simple interface.
- Implement as:
  - Private History
  - A Private Repository
  - A Private Branch
- [*Compare with* Task Branch *for long lived /joint efforts.*]

Wrap Up, Destinations



The SCM Patterns Book

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


DEVELOP YOUR EXPERTISE
Pragmatic Version Control
Pragmatic Version Control
JUnit Recipes
Pragmatic Project Automation
SD BEST PRACTICES
CONFERENCE & EXPO 2005

Other Books of Interest

Pragmatic Version Control Using Subversion

by Mike Mason

Pragmatic Version Control Using CVS

by Andy Hunt & Dave Thomas

JUnit Recipies

by J. B. Rainsberger

Pragmatic Project Automation

by Mike Clark

Other Pointers

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

- steve@berczuk.com

Questions?