Lifecycle Management Starts at Home:

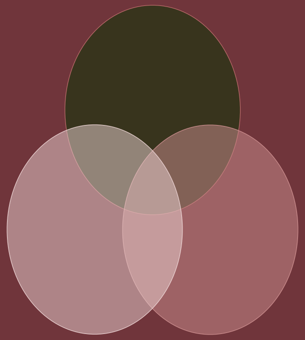
Patterns for Effective Software Configuration Management

Agenda & Goals

- Agenda
  - SCM and The Development Process
  - Agile SCM
  - Codeline and Workspace 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

The Context

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



The Goal: Working software that delivers value.

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

Foundations of a 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
- Integration. Starting in the developer workspace.

Traditional View of SCM

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

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
  - Executable Knowledge over Documented Knowledge. (e.g. “one step” workspace set up.)
- *Customer Collaboration* over Contract Negotiation
  - The codeline is the state of the system. Iterate and change course. Manage expectations.
- *Responding to Change* over Following a Plan
  - SCM is about facilitating change, not preventing it. Feedback through build and test processes.

Effective SCM

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

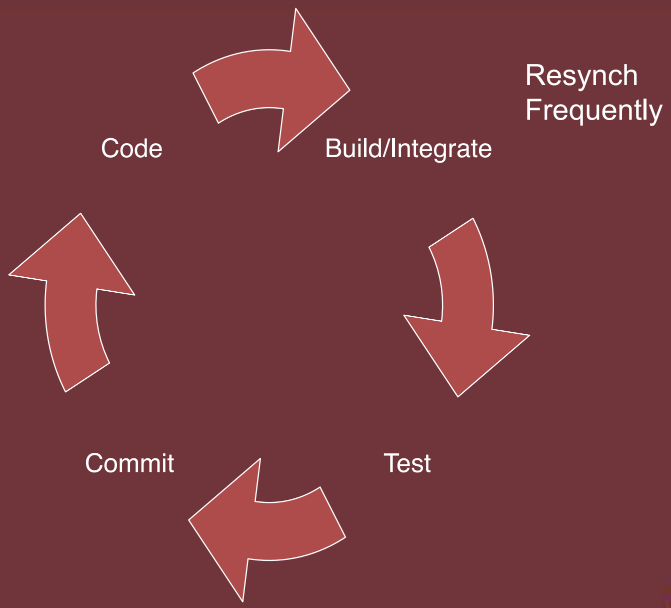
Think about the entire value chain.

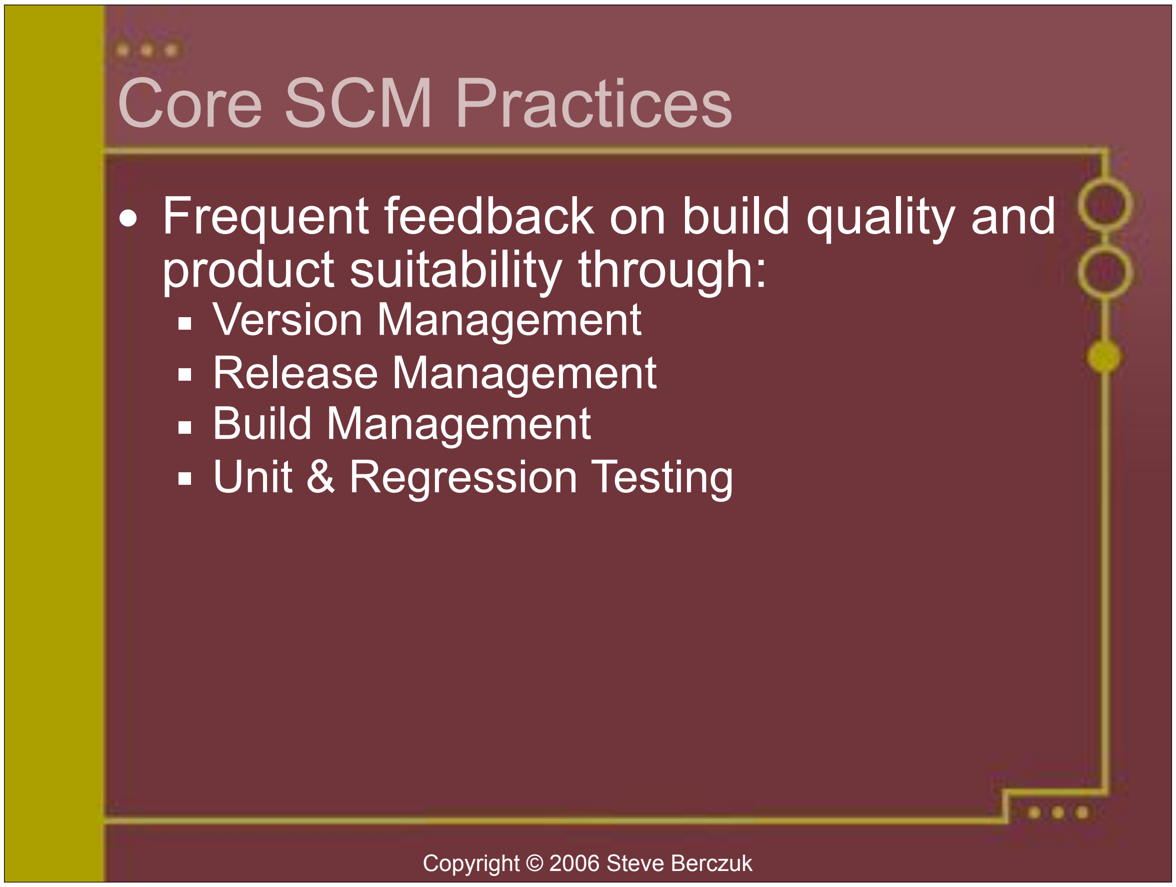
What Agile SCM is Not

- Lack of process
- Chaos
- Lack of control

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

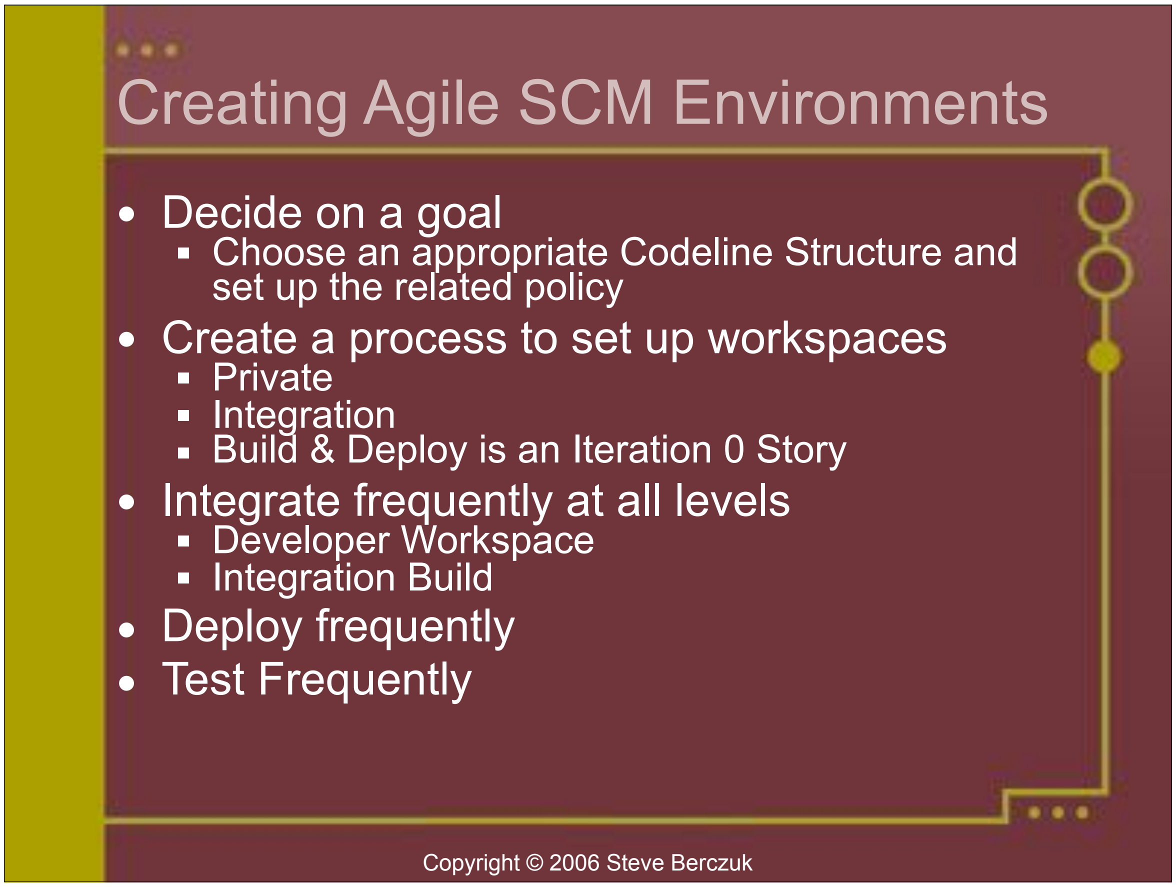
The Agile SCM Cycle





Core SCM Practices

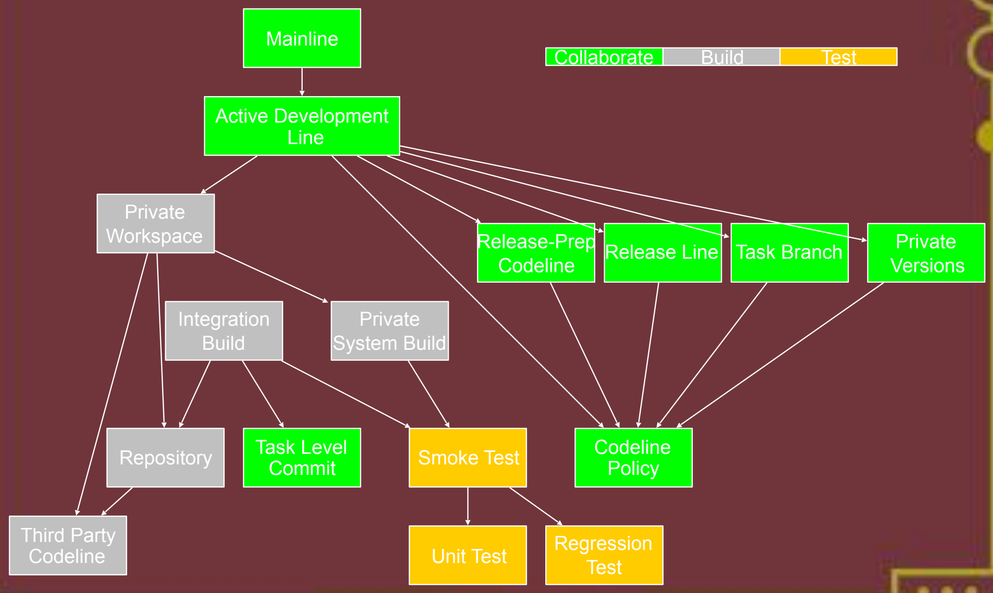
- Frequent feedback on build quality and product suitability through:
  - Version Management
  - Release Management
  - Build Management
  - Unit & Regression Testing



Creating Agile SCM Environments

- 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 Frequently

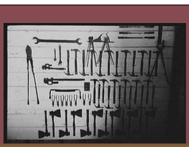
The SCM Pattern Language

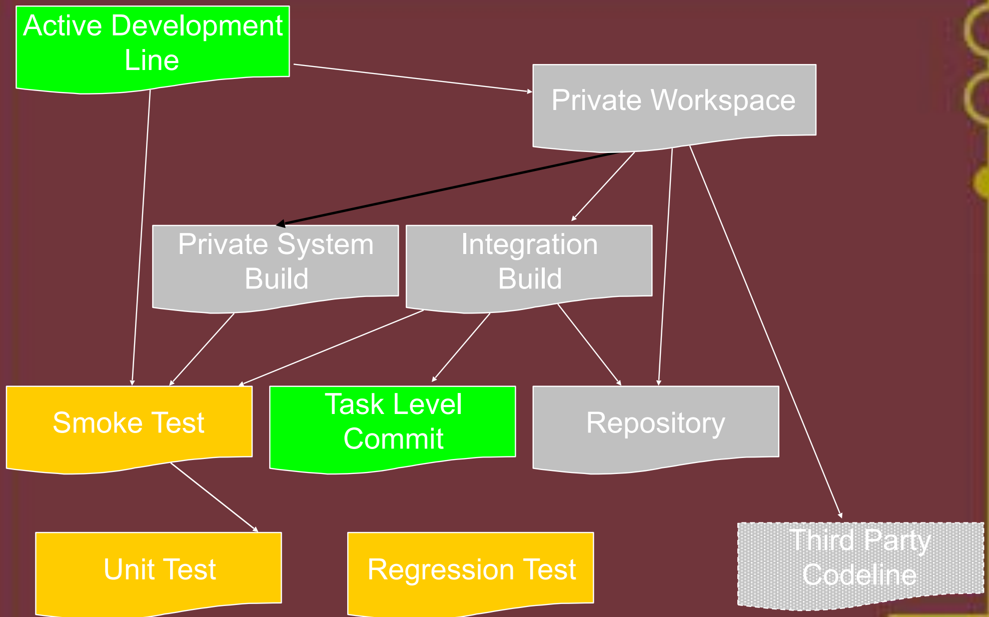


Starting at Home

- Create a Workspace
- Integrate and Build Locally
- Test Locally
- Commit Changes
- Integrate, Build Test in the Integration Workspace

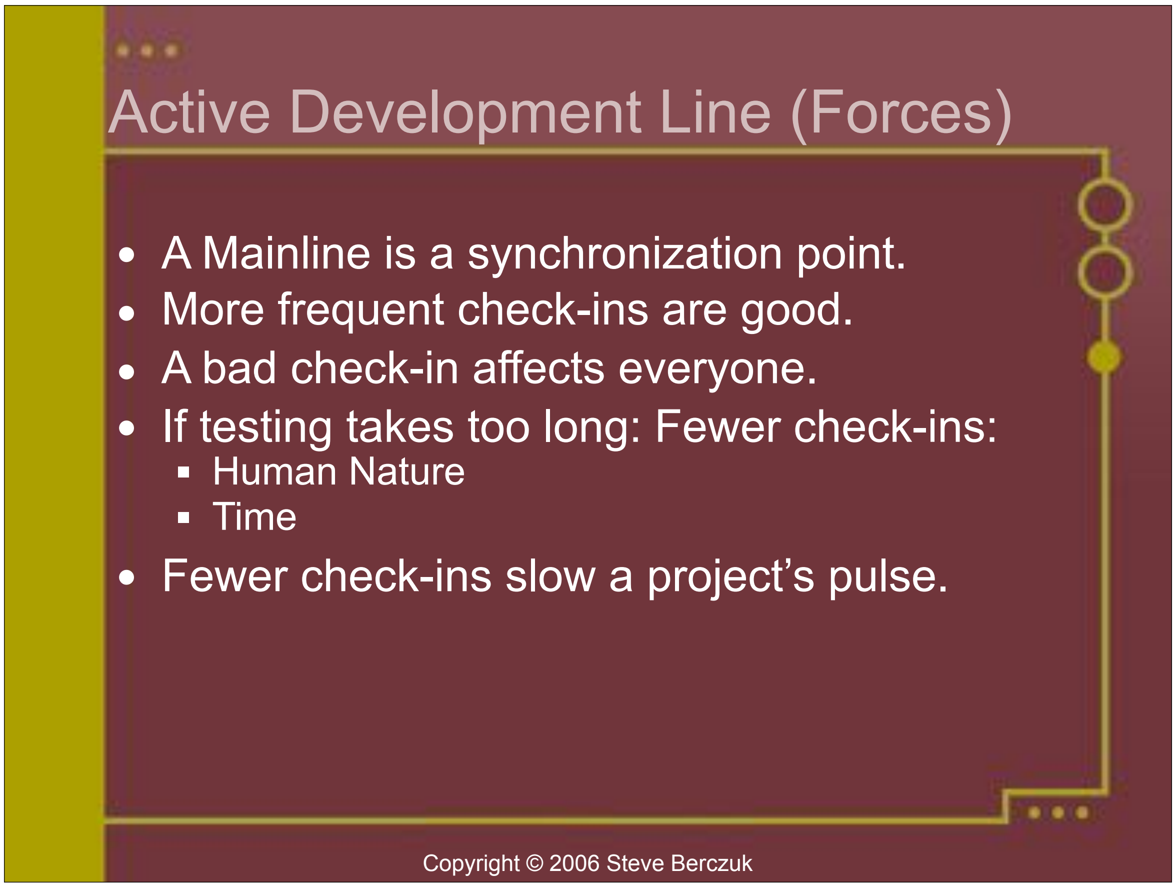
Workspace Patterns





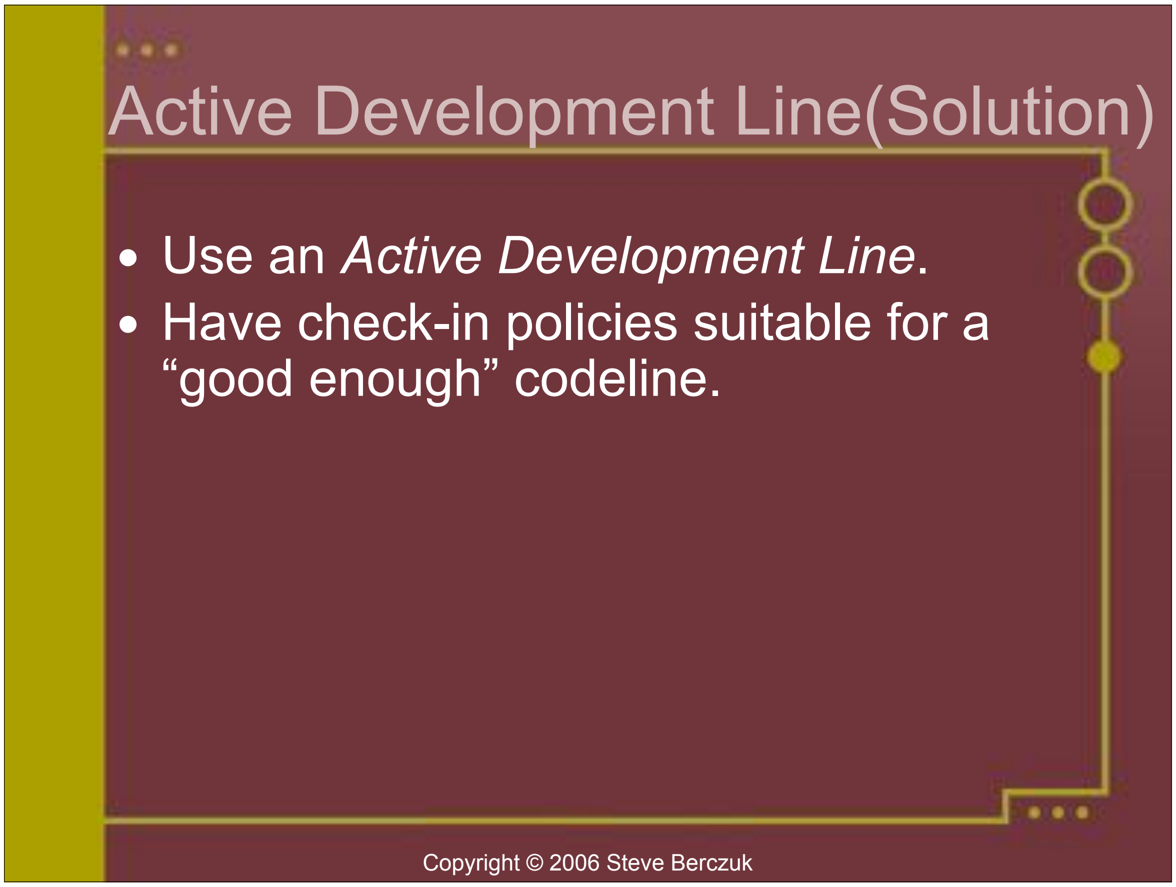
Active Development Line

- 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 slow a project’s pulse.

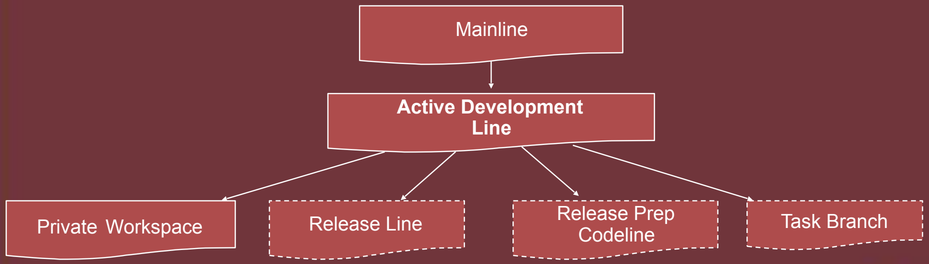


Active Development Line(Solution)

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

Active Development Line (Issues)

- Doing development: *Private Workspace*
- 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)

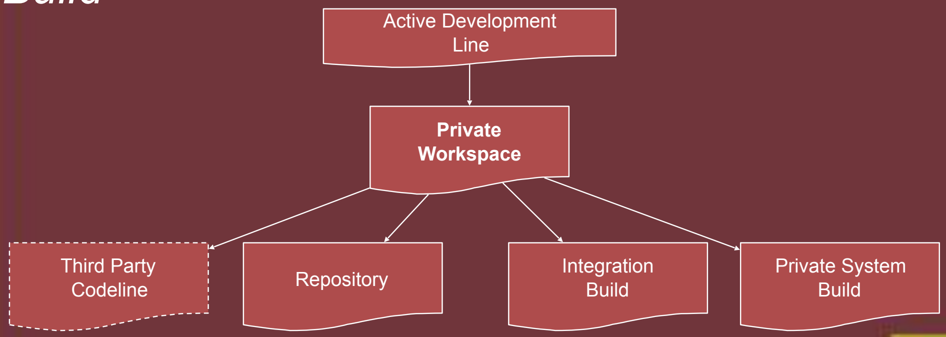
- Frequent integration avoids working with old code.
- People work in discrete steps: Integration can never be "continuous."
- Sometimes you need different code.
- Too much isolation makes life difficult for all.

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*



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)

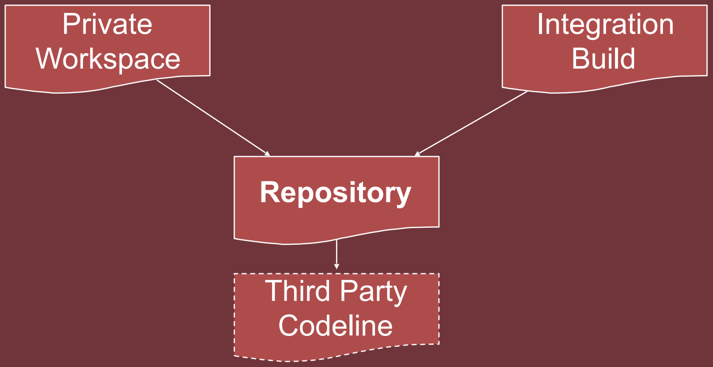
- You want to be able to easily build a workspace from nothing.
- Many things make up a workspace: code, libraries, scripts
- These components could come from a variety of sources (3rd Parties, other groups, etc).

Repository (Solution)

- Have a single point of access for everything.
- Have a mechanism to support easily getting things from the *Repository*.
  - Install VC tools, compiler, etc
  - Check out a project
  - Run a build script.
- Document this process; Briefly
  - "Getting Started" page on a wiki, for example.

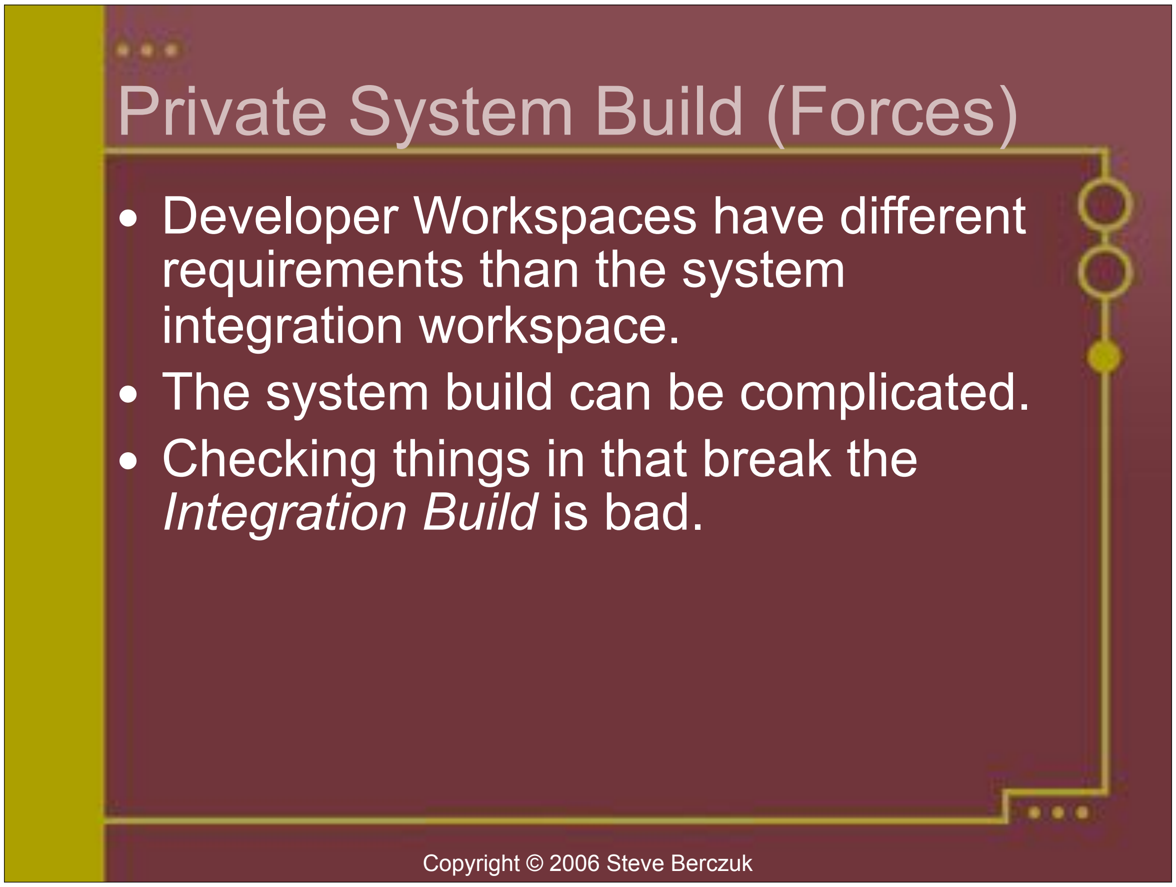
Repository (Unresolved)

- Manage external components: *Third Party Codeline*



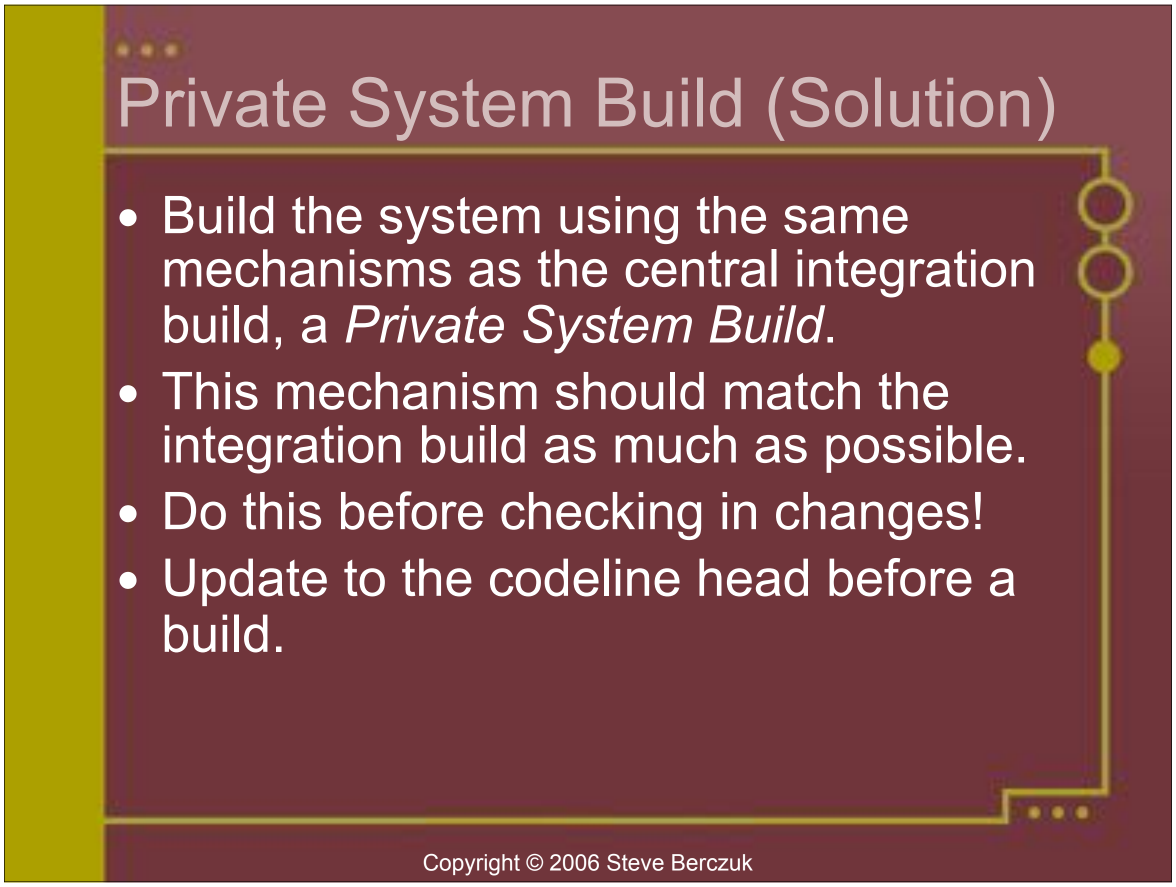
Private System Build

- 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)

- 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.

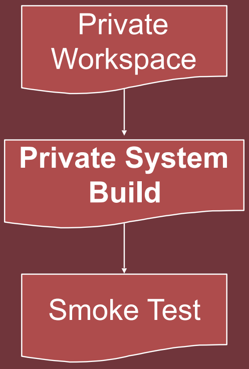


Private System Build (Solution)

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

Private System Build (Issues)

- Testing what you built: *Smoke Test*



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)

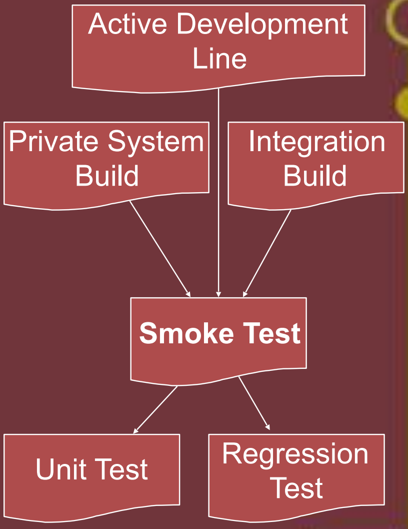
- Exhaustive testing is best for ensuring quality.
- Longer tests imply longer check-ins
  - Less frequent check-ins.
  - Baseline more likely to have moved forward.

Smoke Test (Solution)

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

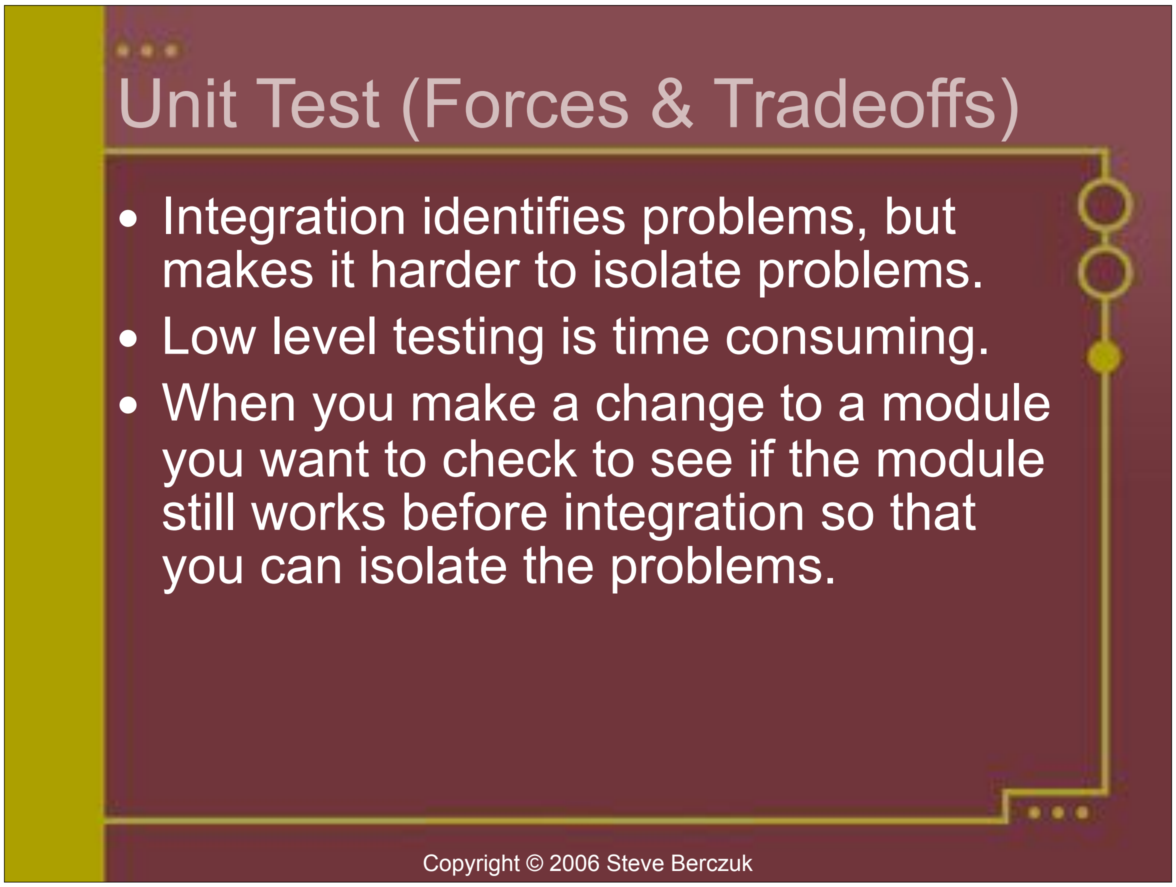
Smoke Test (Unresolved)

- 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*



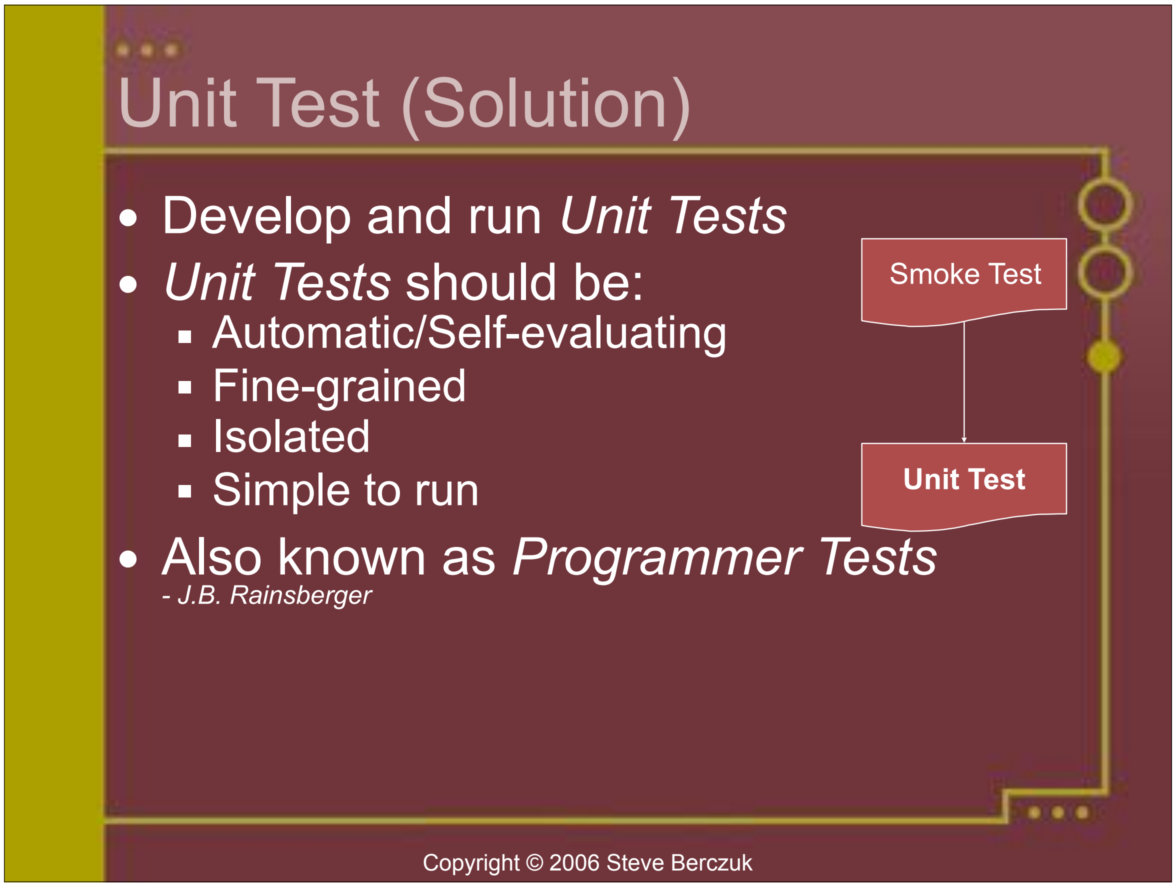
Unit Test

- 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?**



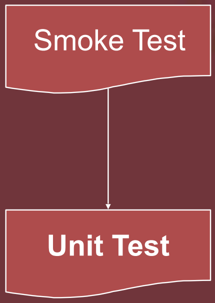
Unit Test (Forces & Tradeoffs)

- 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

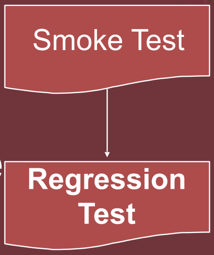
- A *Smoke Test* is good but not comprehensive.
- **How do you ensure that existing code does not get worse after you make changes?**

Regression Test (Forces)

- 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.



Integration Build

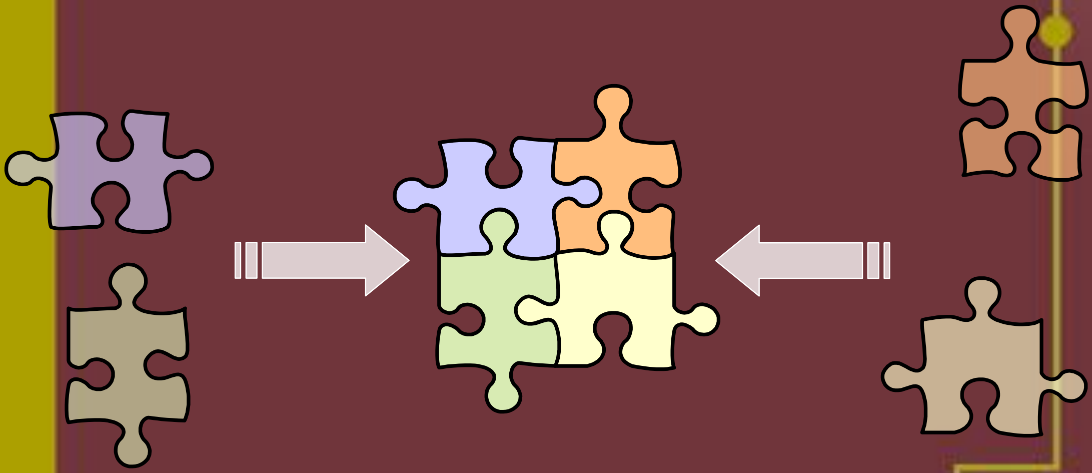
- What is done in a *Private Workspace* must be shared with the world.
- **How do you make sure that the code base always builds reliably?**

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.

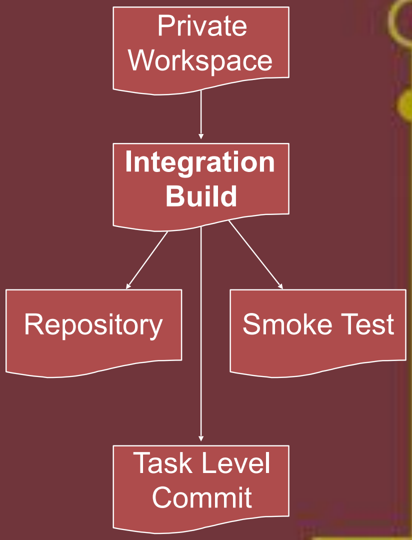
Integration Build (Solution)

- Do a centralized build for the entire code base.



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

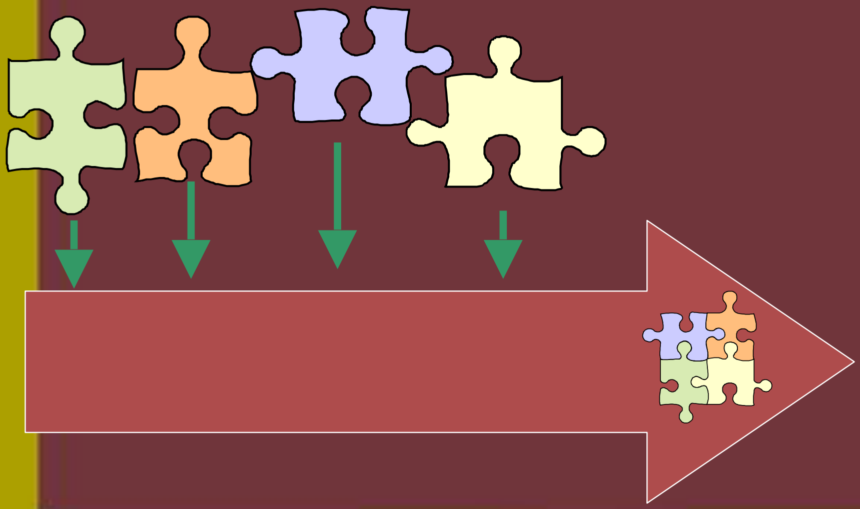
- You need to associate changes with an *Integration Build*.
- **How much work should you do before checking in changes?**

Task Level Commit (Forces)

- 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)

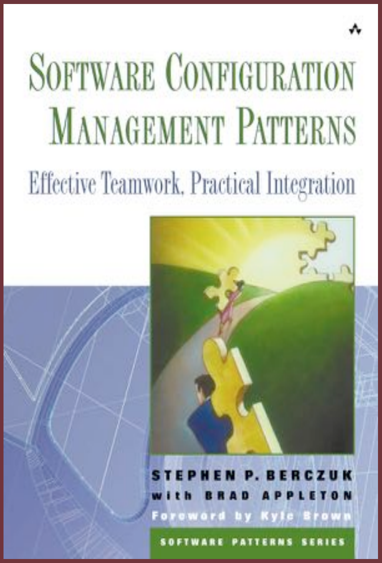
- Do one commit per small-grained task.



The Lifecycle Starts at Home

- Good Developer Workspace Process
- Frequent Integration
- Testing
- Feedback

The SCM Patterns Book


Software Configuration Management Patterns
Effective Teamwork, Practical Integration
Stephen P. Berczuk
with Brad Appleton
Foreword by Kyle Brown
Software Patterns Series

- Pub Nov 2002 By Addison-Wesley Professional.
- ISBN: 0201741172
- Web Sites:
  - www.scmpatterns.com
  - www.berczuk.com
  - www.cmcrossroads.com

Questions?