SCM for Agile Development

Steve Berczuk

About Me

Scrum Practitioner

Overview

- SCM Concepts
- Motivation
- How to Implement Agile SCM (overview)
- Essential SCM Patterns and Practices
- Questions

What: SCM Concepts

- Identification
- Control
- Status Accounting
- Audit & Review

The Role of SCM (Agile Teams)

- Coordination
- Version Management
- Build Management
- Agile teams need to have the right amount of configuration “Management”

5

Value of SCM

- Agile goals:
  - Delivering value
  - Eliminate waste
- SCM enables:
  - Reliable delivery
  - Maintenance & Support
  - Easier change Tracking

6

(Good)

Software Configuration Management

Is Essential for

Agile Software Development

7

Concepts: Workspace

- Everything you need to develop and test
- Source
- Resources
- Developer, Integration, etc

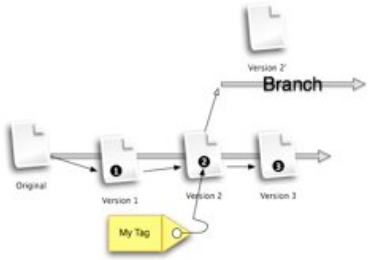
8

Concepts: Codeline

- Definition
- Branch
- Trunk (MAIN)
- Issues:
  - How Many?
  - Enabling Integration
  - Project Rhythm

9

Concepts: Tags, Revisions



10

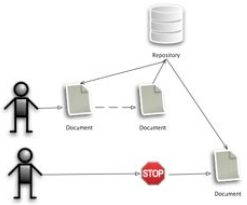
Concepts: Repository

- Version Management System
- SCM
- Examples:
  - Maven Repository
  - Subversion
  - CVS

11

Concepts: Locking Models

- Pessimistic Locking
  - Lock, Modify, Unlock
  - One editor at a time
  - Need to be able to override locks



12

Concepts: Locking Models

- Optimistic Locking
  - Copy, Modify, Merge
  - Allows for concurrency

13

Concepts (Review)

- Features and Value of SCM
- SCM and Version Control
- Codeline & Commit
- Workspace
- Locking Models

14

Why SCM?

15

Common Problems (I)

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

16

Common Problems (II)

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

17

Agility and SCM

- Agile Methods emphasize:
  - Feedback
  - Communication
  - Process that adds value
- Agile SCM
  - Simple and effective SCM
  - Enables development
  - Not only for agile teams
- Balance Feedback and Stability

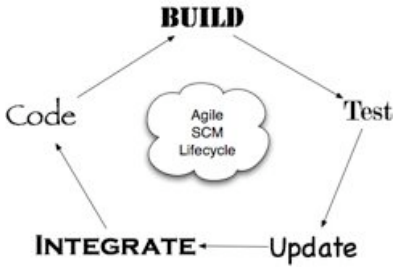
18

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

19

The Agile SCM Cycle



20

What: (Agile)SCM Concepts

- Identification
  - Repository/Build Scripts/Tags
- Control
  - Iteration Planning
- Status Accounting
  - Dashboards
- Audit & Review
  - Unit and Acceptance Testing

21

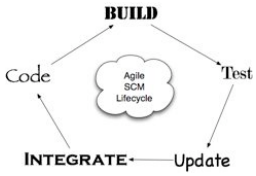
Why? (Review)

- Productivity
  - Save Time
  - Faster Delivery
- Value
- Balance
  - stability
  - progress
- Communication & Coordination

22

How

- Environment
- Practices
- Patterns



23

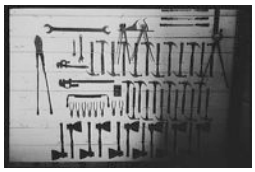
Environment

- Organization
- Architecture
- SCM Process

24

Tools

- Tools Enable
- Process before Tools



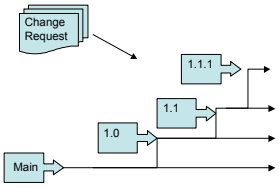
25

Key Development Practices

- Codeline Structure
- Private Workspace
- Build
- Test
- Deploy
- (Automate)

26

Codeline Structure: Too Many Branches?



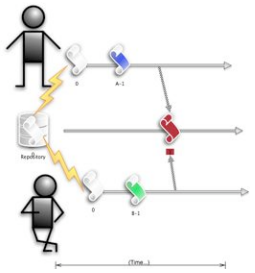
27

Mainline

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

28

Delayed Integration



29

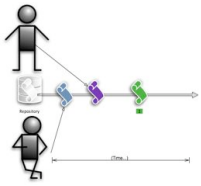
Mainline Tradeoffs

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

30

Mainline (Solution)

- When in doubt, do all of your work off of a single *Mainline*.
- Integrate often.
- Understand why you want to branch, and consider the costs.
- You need to address architecture, code, and tests.



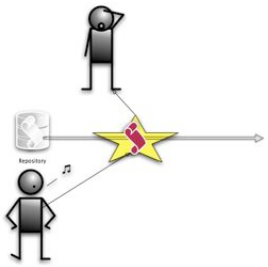
31

Active Development Line

- You are developing on a *Mainline* but worry about stability.
- **How do you keep a rapidly evolving codeline stable enough to be useful without slowing people down?**

32

Fragile Codelines



33

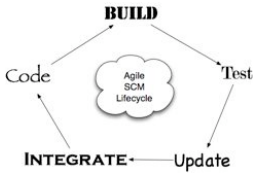
Active Development Line

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

34

Workspaces and Builds

- Support Testing Required for *Active Development Line*.
- New Person Starting on a project
- Code, build, and test
- Commit changes



35

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?**

36

Private Workspace

- Frequent integration
  - Shows problems sooner.
  - Integration problems can disrupt flow.
- Excessive isolation defers problems.
- Shared workspaces can be problematic:
  - Sometimes you need different code.
- Parallel work

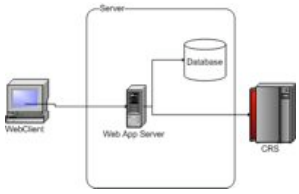
37

Private Workspace

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

38

Private Workspace Example



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

39

Private Workspace Requires

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

40

Repository

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

41

Repository

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

42

Repository

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

43

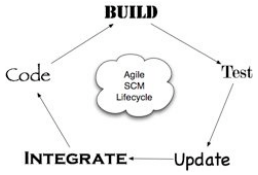
Hierarchy of configuration

- Common Settings
- Environment Specific Settings
  - Development
  - Integration
  - Production
- User-specific overrides
- Can be handled in architecture

44

Builds

- Value *Working software*
- Builds at various levels:
  - Developer
  - Integration
  - Release
- Build scripts are code!
- Build : “Compile & Test”
- Deploy frequently



Private System Build

- You need to build and test 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

- Developer Workspaces have different requirements.
  - The system build can be complicated.
  - Full Testing can be slow but you want to run all of the tests.
- Changes that break the *Integration Build* are bad.
- It can be costly to fix broken builds.

Private System Build

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

Task Level Commit

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

49

Task Level Commit

- Smaller tasks are easier to roll back.
- Large changes mean more isolation.
- A check-in requires some work.
  - Build, Test
  - It is tempting to batch many small changes.
- Issue tracking systems track units of work.
- Frequent Commits provide for safety.

50

Task Level Commit (Solution)

- Do one commit per small-grained task.
- Story, Task, Issue
- Changes include related tests

51

Activity: Build Time Tradeoffs

- Tradeoffs
- More Testing or Faster Build

52

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

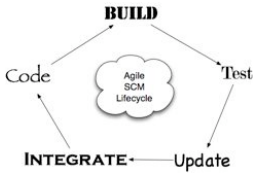
- People work independently.
- *Private System Build*s validate the system.
- Building everything may take a long time.
- Testing everything takes a long time.
- You want to ensure that the codeline works.
- Environmental differences happen
  - Want a canonical definition of "works."

Integration Build

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

Workspaces and Build (Review)

- Single Codeline
- Consistent Workspaces
- Consistent Builds



Types of Tests

57

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?**

58

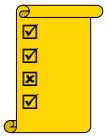
Unit Test

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

59

Unit Test

- Develop and run *Unit Tests*
- Almost nothing is too trivial to test
- *Unit Tests* should be:
  - Automatic/Self-evaluating
  - Fine-grained
  - Isolated
  - Simple to run
- Also known as *Programmer Tests*
  - J.B. Rainsberger



60

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?**

61

Smoke Test

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

62

Smoke Test

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

63

Smoke Test Example

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

64

Integration Test

- End to end test.
- Finds gaps in unit tests.
- During Integration Build.
- Run "feature-level" integration test before commit.
- See: Regression Test.

65

Regression Test

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

66

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.

67

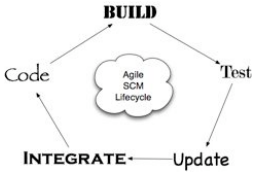
Regression Test

- Develop *Regression Tests* based on:
  - Failed test cases.
  - Problem Reports.
- Run *Regression Tests* whenever you want to validate the system.
- Run these tests as part of an automated build.
  - (nightly or more frequently).

68

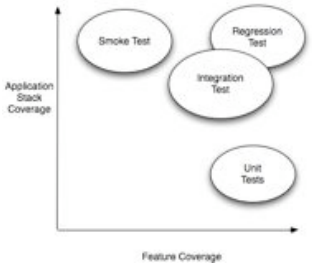
Testing

- Layers of Tests.
- When to Run.
- Balance testing and rate of change.



69

Types of Tests



70

More than one Codeline

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

71

Codeline Policy

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

72

Codeline Policy

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

73

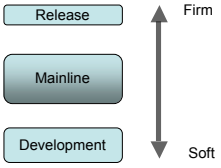
Codeline Policy

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

74

Policies: The Tofu Scale

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



75

Release Line

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

76

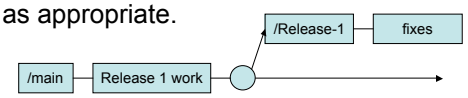
Release Line

- 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.

77

Release Line

- Split maintenance/release activity between
  - *Active Development Line* (New)
  - a *Release Line* (Fixes).
- Propagate changes to Mainline as appropriate.



78

Third Party Codeline

- *Private Workspaces* and the *Repository* need the right versions of external components. You may need to modify third party components.
- **How do you coordinate versions of external components with your versions?**

79

Third Party Codeline

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

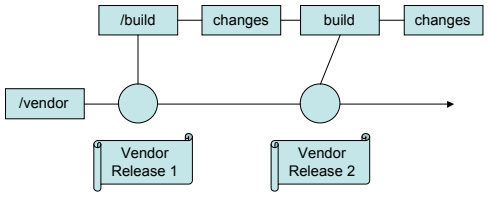
80

Third Party Codeline

- 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.

81

Third Party Codeline (Structure)



82

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?**

83

Task Branch

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

84

Task Branch

- 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.]

85

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?**

86

Private Versions

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

87

Private Versions

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

88

Release Prep Codeline

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

89

Release-Prep Codeline (Forces)

- You want to stabilize a codeline:
  - so you can ship it.
- You want to work on new work during stabilization period.
- A code freeze slows things down.
- Branches have overhead.

90

Release Prep Codeline

- 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.

91

Essential Practices

- Workspace Creation
- Build
- Continuous Integration
- Simple Codelines
- Tests

92

Resources (Web)

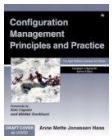
- SCM Patterns Book & Web Site: www.scmpatterns.com
- CM Crossroads: www.cmcrossroads.com
- Brad Appleton's Sites: a
  - acme.bradapp.net
  - Blog.bradapp.net

93

Questions?

94

Resources (Books)



95