SCM Patterns for "Agile" Architectures

Brad Appleton
Software CM/ALM Solution Architect
Arlington Heights, IL
brad@bradapp.net

Agenda

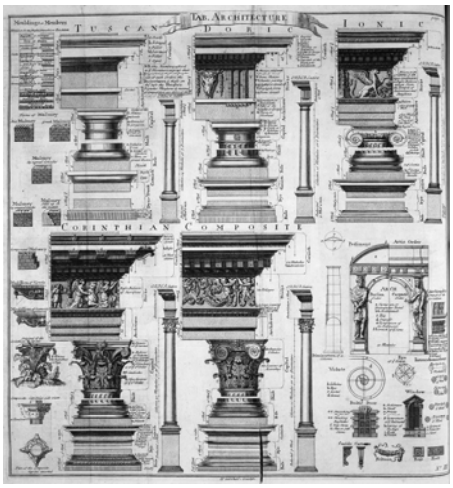
Part I: Background/Foundation

What is “Software Architecture”?

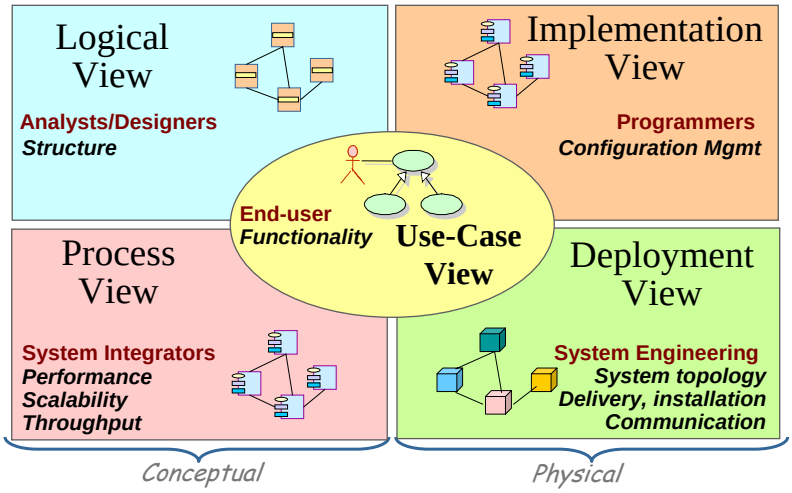
The fundamental organization of a system:

- embodied in its components,
- their relationships to each other and the environment,
- and the principles governing its design and evolution

-- IEEE 1471-2000



4+1 Views model of Software Architecture (Kruchten & UML/RUP)

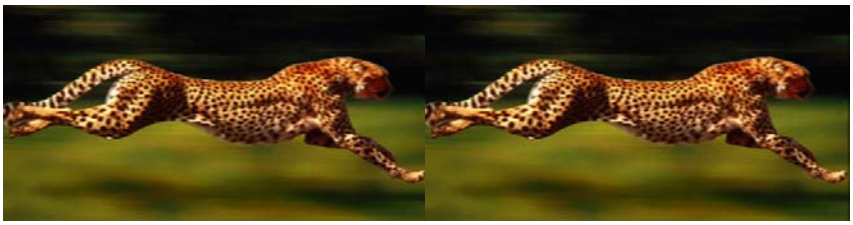


What is Agility?

"The ability to both create and respond to change in order to profit in a turbulent business environment."

-- James Highsmith, *Agile Software Development Ecosystems*

Rapid Response with ***Efficiency*** in Motion, ***Economy*** of Effort, ***Energy*** in Execution, and ***Efficacy*** of Impact!



Agile Development Characteristics

Adaptive – responsive to change in needs & requirements via continuous feedback and reflective retrospection

Goal-driven – focus on producing executable-results (working functionality) in order of highest business value.

Iterative – short development cycles, frequent releases, regular feedback

Lean – simple design, streamlined processes, elimination of redundant information, minimal intermediate artifacts

Emergent behavior – highly collaborative self-organizing teams in close interaction with stakeholders

Dr.Dobb's ARCHITECTURE & DESIGN WORLD DESIGN, MODEL AND BUILD SOLID FRAMEWORKS JULY 17-20, 2006, CHICAGO, IL

Principles of Lean Development

- Eliminate Waste *(Minimize Artifacts & Add Nothing but Value)*
- Build Quality In *(Satisfy All Stakeholders & Deploy Comprehensive Testing)*
- Amplify Learning *(Learn by Experimentation)*
- Defer Commitment *(Decide as Late as Possible)*
- Deliver Fast *(Deliver as Fast as Possible)*
- Respect People *(Decide as Low as Possible)*
- Optimize the "Whole" *(Measure Business Impact & Optimize Across Organizations)*

Source: Mary & Tom Poppendieck, http://www.poppendieck.com/

Theory of Constraints – 5 Focusing Steps

1. IDENTIFY the Constraint
2. EXPLOIT the Constraint
3. SUBORDINATE to the Constraint
4. ELEVATE the Constraint
5. Repeat – PREVENT INERTIA from becoming the Constraint

Dr.Dobb's ARCHITECTURE & DESIGN WORLD DESIGN, MODEL AND BUILD SOLID FRAMEWORKS JULY 17-20, 2006, CHICAGO, IL

What is an "Agile Architecture"?

- *Gracefully evolves & adapts* to meet changing needs & constraints
- *Resilient & responsive* to change

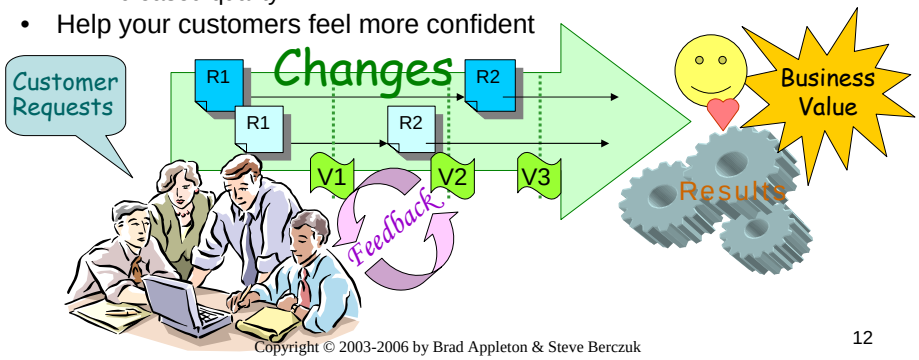
What is SCM? (Traditional view)

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

What is SCM? (Agile view)

SCM is a set of structures & practices that:

- Facilitate frequent feedback on build quality & product suitability
- Enable changing & building systems in repeatable, agile fashion with:
  - Increased productivity
  - Enhanced responsiveness to customers
  - Increased quality
- Help your customers feel more confident



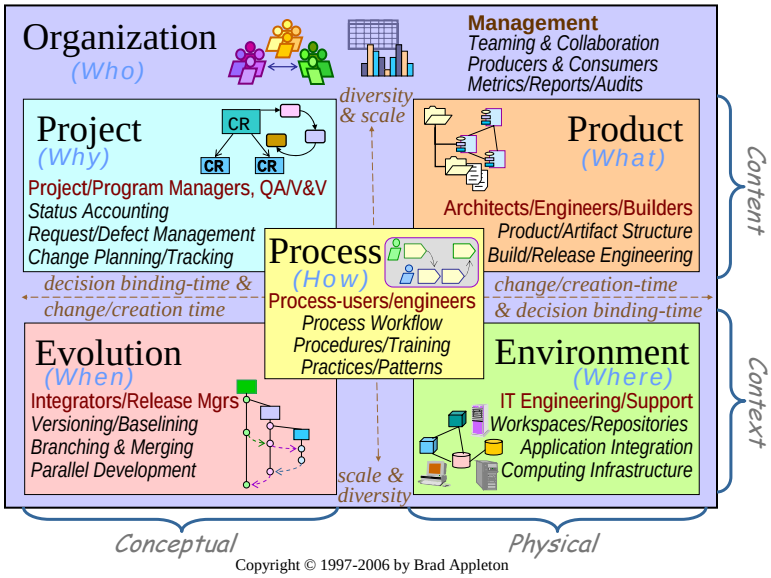
What is SCM? (Architectural view)

Software Configuration Management is the architecture of the evolution of architecture!

4+2 views of SCM "Architecture"

1) The *Project* structures
2) The flow of *Evolution*
3) *Product* Objects/Artifacts
4) Your *Environment*
+1) Your *Processes*
+2) Your *Organization*

4+2 Views of SCM Architecture

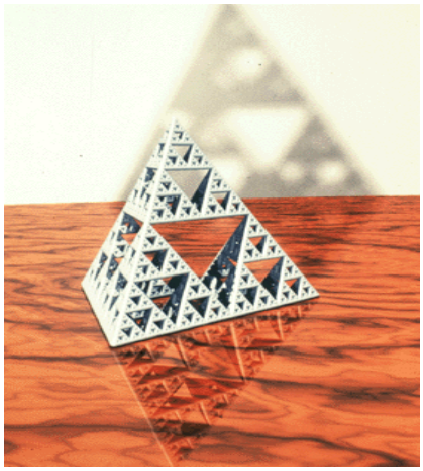


Core SCM Patterns Concepts

- What are Patterns?
- Codeline/Branch
- Configuration
  - Version
  - Revision
  - Label
- Workspace

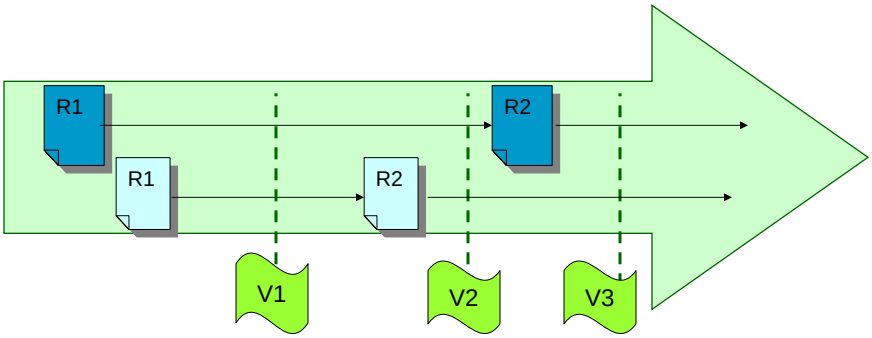
What are Patterns and Pattern Languages?

- 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.



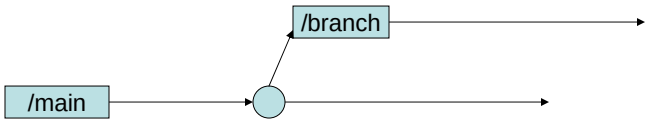
Codeline

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



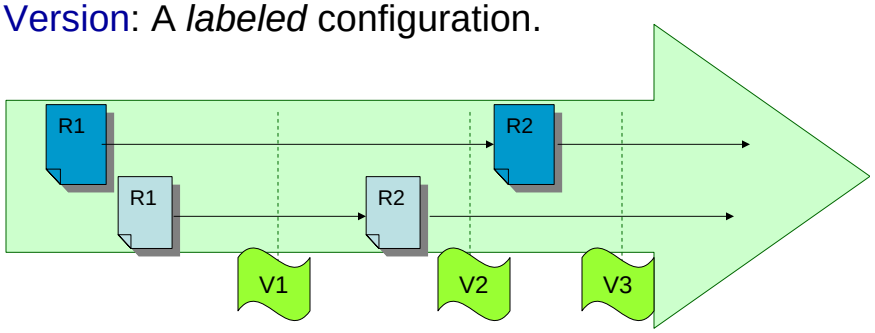
Branching

- **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.



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.



19

Workspace

- Everything you need to build the Product
  - Code, Scripts, Database resources, etc.

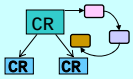
20

Part II: The Patterns

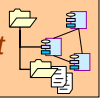
- Codeline Patterns
- Build/Integration Patterns
- Promotion "Leveling" Patterns
- Variability Management
- Wrap-Up

SCM Patterns across the 4 Views

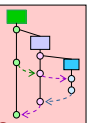
Project
- *Task-Based Development*
- *Promotion Leveling*



Product
- *Private Build*
- *Integration Build*
- *Private Checkpoint*
- *Workspace Update*
- *Smoke/Regression Test*
- *Variability Mgmt*



Evolution
- *Mainline*
- *Release Line*
- *Active Development Line*
- *Release-Prep Codeline*
- *Task Branch*
- *Private Branch*



Environment
- *Private Workspace*
- *Integration Workspace*
- *Staging Area*

Codeline Patterns

- Mainline
- Active Development Line
- Codeline Policy
- Release Line
- Release-Prep Codeline
- Task Branch
- Private Branch

Dr.Dobb's ARCHITECTURE & DESIGN WORLD DESIGN, MODEL AND BUILD SOLID FRAMEWORKS JULY 17-20, 2006, CHICAGO, IL

Codeline Structures for Agility

- 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)?**

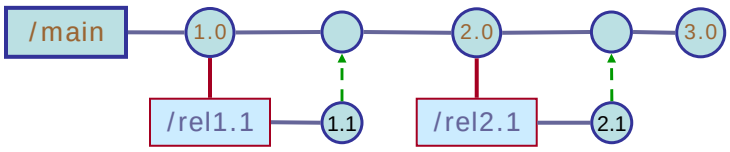
Mainline (Forces & Tradeoffs)

- 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 cause by deferred integration.

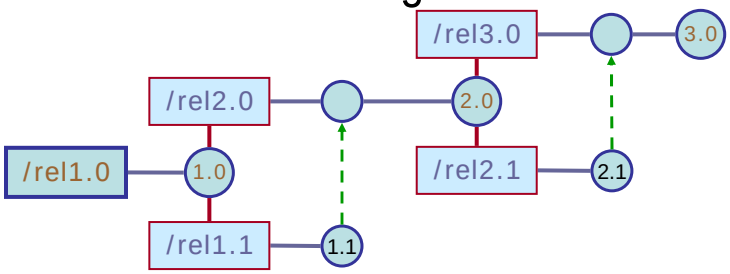
Mainline (Solution)

- Keep *Latest* release/project efforts on Mainline
- Branch *Late/Lazy* to support & maintain previous releases *[use nested synchronization]*
  - ☹ **DON'T *cascade* new branches** for follow-on projects/releases *[avoid staircase branching]*
  - ☺ **DO *sync-merge* to Mainline** ("mainlining") to reduce breadth of branch tree

Mainline



vs. Cascading Staircase



Active Development Line

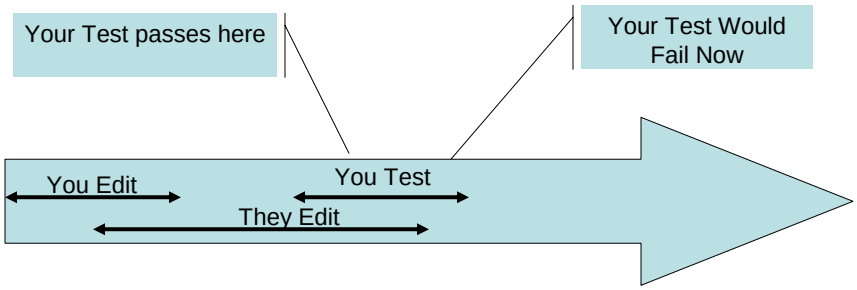
- You are developing on a *Mainline*.
- **How do you keep a rapidly evolving codeline stable enough to be useful (but not impede progress)?**

Active 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.
- *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*

Codeline Policy

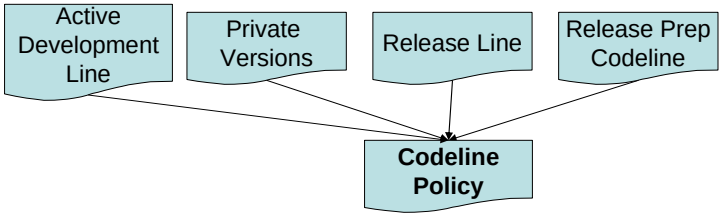
- *Active Development Line* and *Release Line* (etc.) need to have different rules.
- **How do developers know how and when to use each codeline?**

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.



Release Line

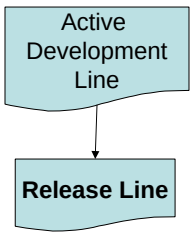
- You want to maintain an *Active Development Line*.
- **How do you do maintenance on a released version without interfering with current work?**

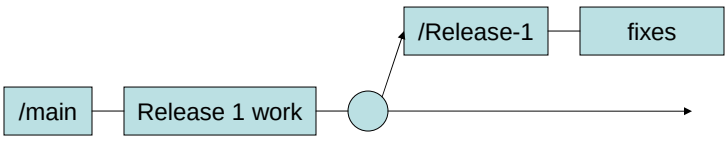
Release Line (Forces & Tradeoffs)

- 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?**

39

Dr.Dobb's ARCHITECTURE & DESIGN WORLD DESIGN, MODEL AND BUILD SOLID FRAMEWORKS JULY 17-20, 2006, CHICAGO, IL

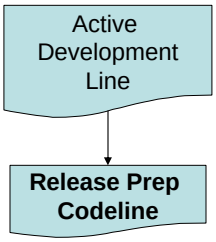
Release-Prep Codeline (Forces & Tradeoffs)

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

40

Release Prep Codeline (Solution)

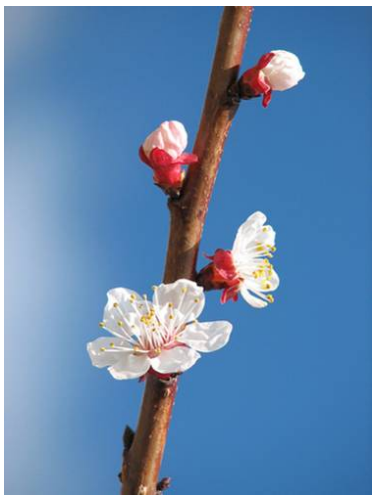
- Branch instead of freeze. Create a *Release Prep Codeline* (a branch) when code is approaching release quality.
- Leave the *Mainline* for active development.
- The *Release Prep Codeline* becomes the *Release Line* (with a stricter policy)
- *Note:* If only a few people are doing work on the next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



Task Branch

Created exclusively for the duration of a single development task

- Good for risky or experimental efforts



Private Branch

Created exclusively for a single developer (or two) for the duration of a project
- Encompasses multiple (sequential) change-tasks
- Good for implementing ***Private Checkpoints***

Build/Integration Patterns

- Private Checkpoint
- Workspace Update
- Task-Level Commit
- Private Build
- Integration Build
- The Three Builds
- Smoke Test
- Regression Test

Private Checkpoint

- 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 Checkpoint (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 Checkpoint (Solution)

- Provide developers with a mechanism for checkpointing changes using a simple interface.
- Can Implement as any of the following:
  - Private Archive/Repository
  - Private Branch
  - Task Branch
  - Private Label/Tag

47

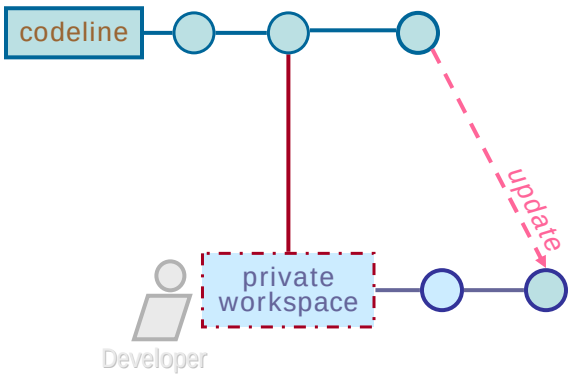
Dr.Dobb's ARCHITECTURE & DESIGN WORLD DESIGN, MODEL AND BUILD SOLID FRAMEWORKS JULY 17-20, 2006, CHICAGO, IL

Workspace Update

- **Synchronize your workspace with the codeline**, without breaking the codeline
- Reconcile recent changes together sooner & keep developers aware of others activities

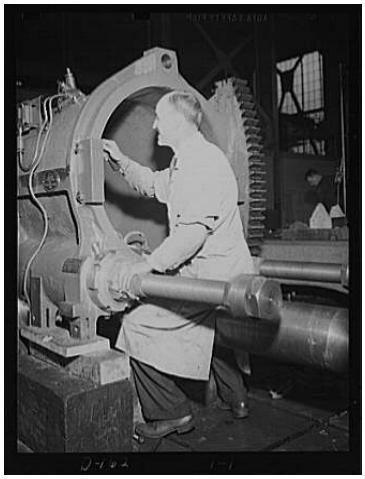
48

Workspace Update



Task Level Commit

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



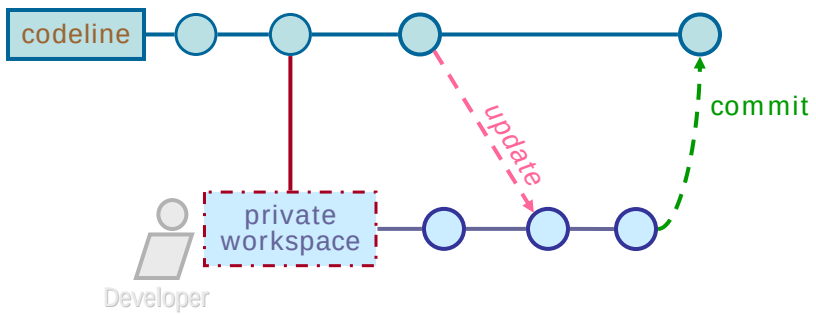
Task Level Commit (Forces & Tradeoffs)

- 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)

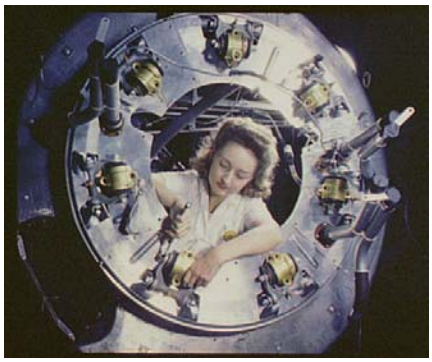
Do one commit per small-grained task.

- [*Compare with* Task Branch *for long lived efforts*]



Private Development 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 Development Build (Forces & Tradeoffs)

- Developer Workspaces have different needs than the system build.
- The system build can be complicated.
- Checking-in changes that break the *Integration Build* is bad.

Private Build (Solution)

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

Integration Build

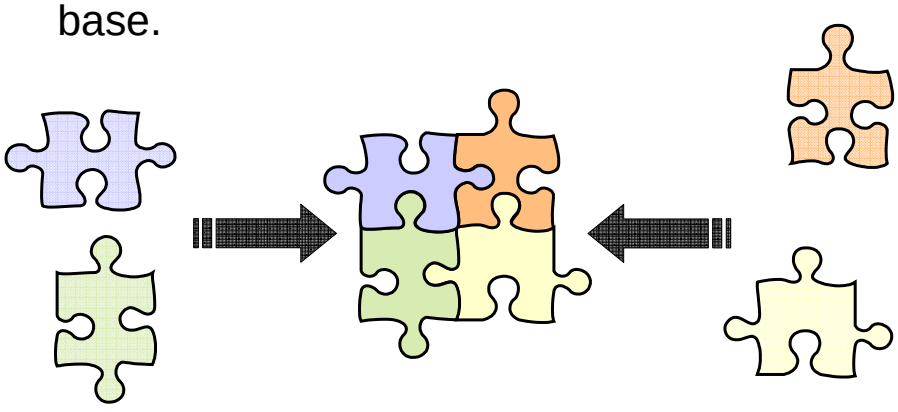
- What is done in a *Private Workspace* must be shared with the world.
- **How do you make sure that the code base always builds reliably?**

Integration Build (Forces & Tradeoffs)

- People do work independently.
- *Private Development 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)

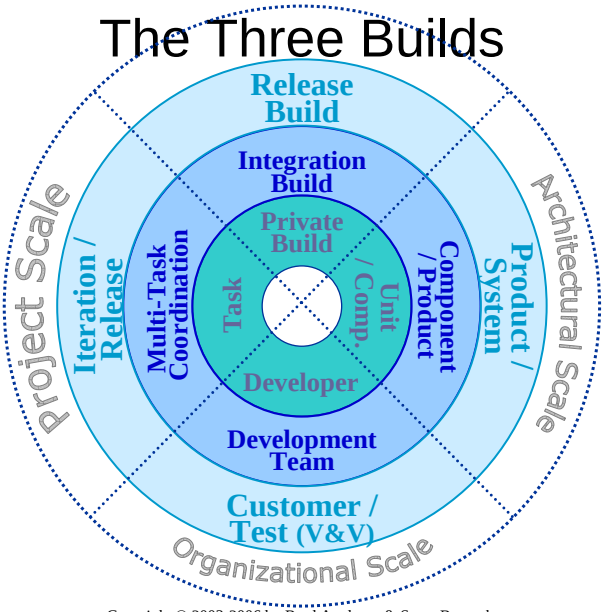
- Do a centralized build for the entire code base.



The Three Builds

- **Private Development Build**
  - Provides a consistent way for the developer to build the software in the confines of their private workspace
- **Team Integration Build**
  - Synchronize team, feedback on code quality/integrity
- **Formal Release Build**
  - Creates the deployable package
- *Why?:*
  - Productivity, predictability, documented, ability to delegate build activity without compromising CM or quality.

The Three Builds



Smoke Test

- You need to verify an *Integration Build* or a *Private 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, encouraging less frequent check-ins.

Smoke Test (Solution)

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

Regression Test

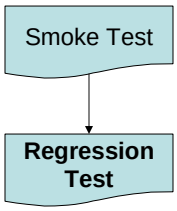
- A *Smoke Test* is good but not comprehensive.
- **How do you ensure that existing code does not get worse after you make changes?**

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.



Promotion "Leveling" Patterns

- What is "Promotion"?
- Version Promotion
- Promotion Workspaces
- Branch Promotion
- Promotion Branches
- Label Promotion

What is a Promotion Lifecycle?

- A series of stages/levels that our end-result must pass through before we are willing to "release" it to others
- A sequence of significant milestone events, each of which represents either:
  - an increase in confidence, *or …*
  - a transfer of responsibility

  in assuring the release-quality of a deliverable

Example Promotion Lifecycles:

- *{Developed, Reviewed, Tested, Audited, Released}*
- *{Development, Staged, Tested, Validated, Production}*
- *{Private, Team, QA, Customer, Failed}*

Common Promotion Mechanisms

- *Version Promotion* (Promoted Versions)
- *Promotion Workspaces*
- *Branch Promotion* (Promoted Branch)
- *Promotion Branches* (Promotion Branching)
- *Label Promotion* (Promoted Label)
- *Promotion Labels* (Promotion Labeling)
  - Equivalent to Version Promotion using Labels as the "attribute values" for the promotion-level

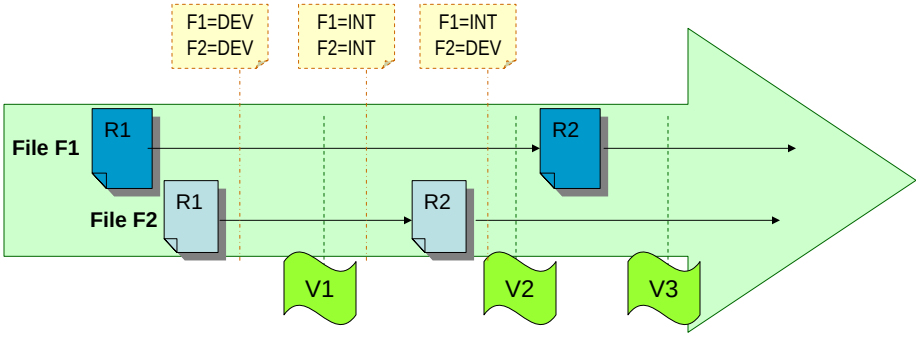
Version Promotion

- The most recent version of each file on the codeline is associated with a promotion level
- File versions are "promoted" to the next level by "advancing" their promotion-level attribute
- Whenever a file is "updated", it starts over again at the initial promotion level.

PRO: Can easily discern if all files on the tip of the codeline are at the correct promotion level, and which files aren't

CON: Can be very cumbersome to implement if you have to do it yourself

Version Promotion



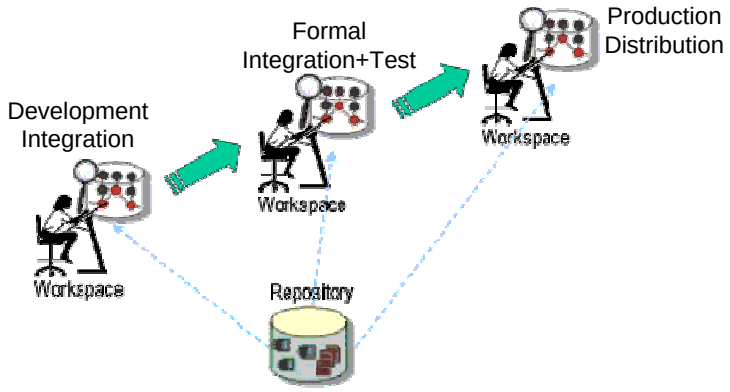
Promotion Workspaces

- Uses a separate workspace or "staging area" for housing built deliverables deliverables
- When a build progresses from one level to the next, the built results are "pushed" to the next-level workspace
- Common example uses three "vaults" (staging areas):
  1. Development Integration "Vault"
  2. Formal Integration+Test "Vault"
  3. Production Distribution "Vault" (Release-Area)

PRO: No confusion over which versions in the workspace are at which "level"

CON: Can be time-consuming to copy/link file versions across workspaces or staging areas

Promotion Workspaces



Promotion Workspaces

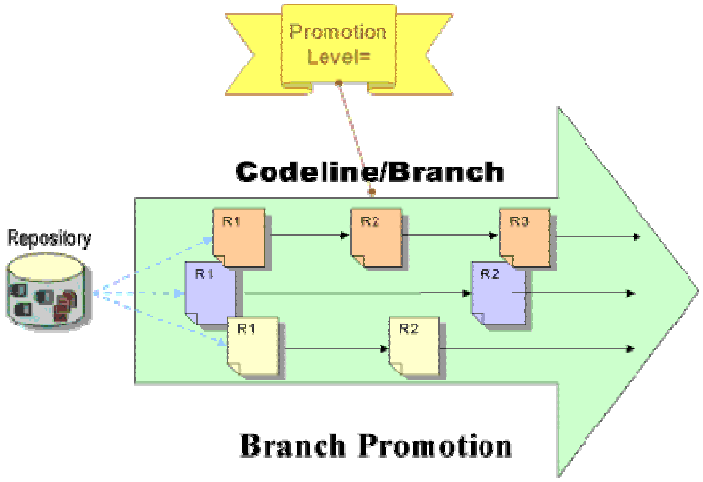
Branch Promotion

- Associate promotion-level attribute with an entire codeline
- When tip of the codeline progresses from one level to the next, advance the branch's promotion-level

PRO: Very quick & easy to "promote" – no files need to be copied/linked

CON: Okay for handing off an entire branch but not as useful when using the same codeline for frequent handoffs (can't tell status of previous handoff anymore)

Branch Promotion



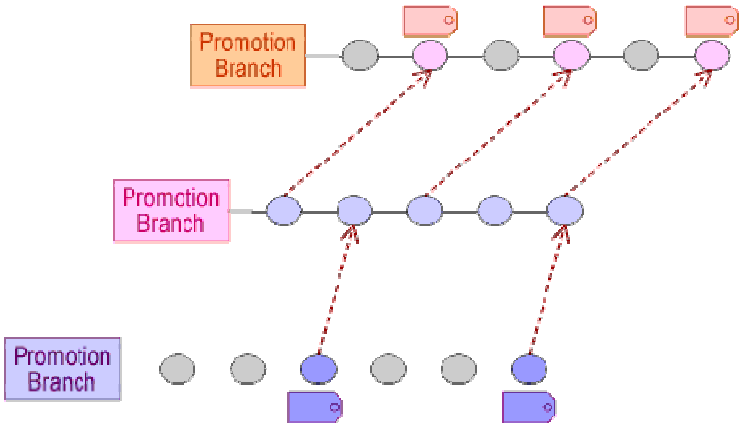
Promotion Branches

- Uses separate branch/codeline for each promotion level
- When a build progresses from one level to the next, the versions are "pushed" (copymerged)
- Very similar to promotion workspaces, but with codelines instead (or in addition)

PRO: Codelines make for nice "integration territories" when transferring responsibility – avoids "turf wars" and "policy disputes" from competing groups by giving each their own codeline and codeline policy

CON: Creates a new version when promoting to the next level (even if no changes were needed)

Promotion Branching



Promotion Branching

77

Dr.Dobb's ARCHITECTURE & DESIGN WORLD DESIGN, MODEL AND BUILD SOLID FRAMEWORKS JULY 17-20, 2006, CHICAGO, IL

Label Promotion

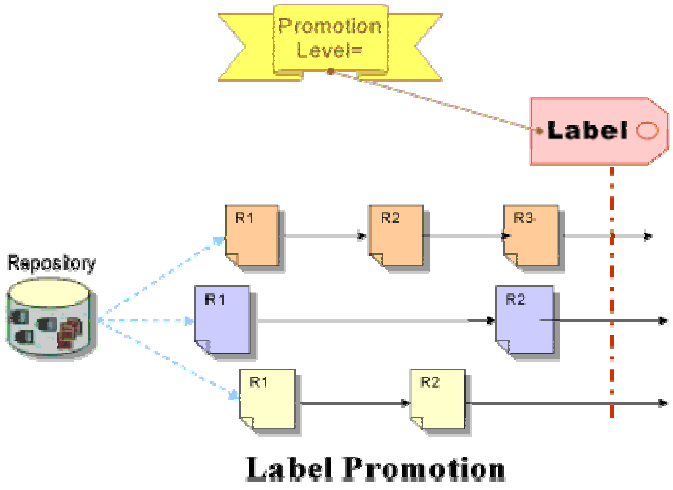
- Associate promotion-level attribute with each label/build
- When build progresses from one level to the next, advance the label's promotion-level

PRO: Very quick & easy to "promote" – no files need to be copied/linked and no versions to merge; allows for multiple builds on the same codeline to each go thru their own promotion levels independently

CON: Can be somewhat unwieldy to implement if your tool doesn't readily support "attributes" on a label/tag

78

Label Promotion



Variability Management

- What is Product Variability?
- Why Product Variability?
- Branching-time or Binding-time?
- Binding Times
- Build/Package Options
- Feature Configuration
- Business Rules
- Composition, Inheritance & Aspects

What is Product Variability?

Product-Lines & Product-Families:

- Variability of a single codebase across *multiple* products:

Multi-Variant Product:

- Variability of a single codebase across *the same* product:

Multiple Products & Multiple Variants

Product-Lines & Product-Families:

- An entire line/family of different products with some "core" (shared) components & functionality
- Each product has some unique combination of additional functionality and/or functional "variation"

Multi-Variant Product:

- Many (functional) variations of the same (product) theme
- Variations are often customer/market-specific
- Different from supporting legacy releases (multi-project):
  - because the functional differences aren't separated by time, but by market/customer and/or technology/platform.

Why Product Variability?

A Company may offer product variability because they may believe …

- "One size does *not* fit all!"
- It will improve competitive advantage
- It will increase their market size/share
- It will uniquely differentiation them in the market (branding)
- Etc.,

Branching-time or Binding-time?

- Many attempt product-variability with Branching by using a codeline per variant; *Don't do this!*
  - Branching is for isolation/synchronization of code across people & places during the same time-period:
    - Concurrent/parallel & remote/distributed development
    - Maintaining legacy/historical versions of an install-base

Branching-time or Binding-time?

Address Variability using late-binding

(instead of branching) **whenever possible!**

– *Branching for Variability is like Cut-n-Paste Reuse!*

- Creates multiple instances of the same code that all need to be repaired for the same "bugfix" or enhancement
- Creates more merging & integration for something that is fundamentally *not* an issue of isolation+synchronization

Binding-Times

Example decision *binding times* for a point of functional variation (*variation point*):

- ***Source reuse time*** - when reusing a configurable source artifact
- ***Development time***
- ***Static code instantiation time*** - during generation/assembly of code for build
- **Build time** - during compilation or related processing

Source: www.softwareproductlines.com

More Binding-Times

- **Package time** - when assembling binary & executable collections
- ***Customer customizations***. Decisions bound during custom coding at customer site
- ***Install/Upgrade time***. Decisions bound during the installation of the software product
- ***Startup time***. Decisions bound during system startup
- ***Runtime***. Decisions bound when the system is executing

Source: www.softwareproductlines.com

Build/Package Options

Manage platform differences with well known design & architecture patterns that "bind" at build-time or package-time:

- ***Wrapper-Façade*** *[POSA2]*
- Numerous patterns from the Gang-of-Four design patterns book (***Factory***, ***Factory Method***, ***Bridge***, etc.)
- Combine with Make/ANT options & variables and judicious use of conditional compilation

Post-Release Feature Configuration

Manage variation in features/feature-sets with selection & deselection patterns:

- Enables/disable features and services at post-release binding-times (install/upgrade-time & run-time)
- ***Component Configurator, Interceptor, Extension Interface*** *[POSA2]*
- Variations of ***Register/Unregister*** and ***Publish-Subscribe*** in a feature/service "registry"

89

Dr.Dobb's ARCHITECTURE & DESIGN WORLD DESIGN, MODEL AND BUILD SOLID FRAMEWORKS JULY 17-20, 2006, CHICAGO, IL

Business Rules

Differences in policy/mechanism may be handled using a business-rules approach

- Maintains a single codebase + codeline to deliver a single product with multiple possible configurations of rules and rule-settings
- ***Strategy, Template Method, Adapter, Decorator*** *["Gang of Four"]*
- ***Adaptive Object-Model*** *[Yoder & Johnson]*
- ***Application "resource settings"***

90

Composition, Inheritance & Aspects

- Composition & Delegation are usually best
- Inheritance may be useful in some cases
  - if the type of configuration needed really does fit a single hierarchical model of increasing specialization
- In other cases, an aspect-oriented approach might be better
  - if the "seams" of configurability cut-across multiple components/services

Dr.Dobb's ARCHITECTURE & DESIGN WORLD DESIGN, MODEL AND BUILD SOLID FRAMEWORKS JULY 17-20, 2006, CHICAGO, IL

Wrap Up

- Lean Branching
- CM Constraints
- Promotion Notions
- SCM Patterns Book
- Managing Multiple Variants/Products
- Other "Agile SCM" Resources

Principles of "Lean" Branching

- Deliver as fast as possible
  - Integrate fine-grained change-tasks as early as possible
- Decide as late as possible
  - Branch as late as possible
- Decide as low as possible
  - Developers reconcile merges and commit their own changes
- Optimize across the "Whole"
  - Use a Mainline to maintain a manageable branching structure

93

Dr.Dobb's ARCHITECTURE & DESIGN WORLD DESIGN, MODEL AND BUILD SOLID FRAMEWORKS JULY 17-20, 2006, CHICAGO, IL

Eliminate CM Constraints

Remove Integration/Build/Test "Bottlenecks"

- One of the single biggest "drags" on development feedback cycle-time is the "friction" that comes from prohibitive build-times, or long testing-cycles
- These force development to either freeze or branch the code-base for significant periods of time while waiting for integration/build/test activities to complete
- Integration+Build tools/scripts, code structure, and network resources must be leveraged appropriately to minimize build times

94

Applicability of Promotion Mechanisms

- Can mix & match mechanisms as appropriate
  - *Branch Promotion*: useful when the branch is the unit of handoff or when handoffs are infrequent on that branch
  - *Promotion Branching*: useful for separate levels of integration (so merging would be performed anyway) or separate owners/policies
  - *Label Promotion*: well suited for a promotion levels at the same level (scope) of integration.
- Be wary of Branching/Labeling for promotion purposes if wouldn't otherwise make sense to branch/label or merge
- Don't "force-fit"! Some of these things emerge "naturally"
  - Private/Task Branch ⇒ Active Line ⇒ Release Line
  - Private Build ⇒ Integration Build ⇒ QA/Release Build

95

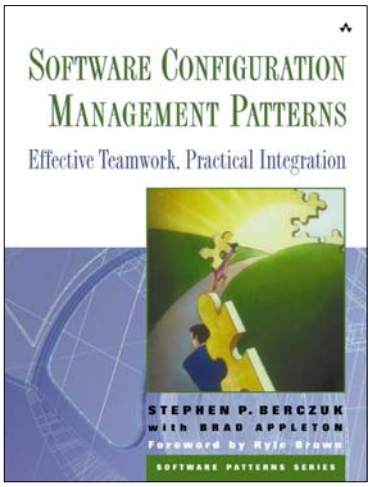
Dr.Dobb's ARCHITECTURE & DESIGN WORLD DESIGN, MODEL AND BUILD SOLID FRAMEWORKS JULY 17-20, 2006, CHICAGO, IL

Managing Product Variability

- Use Late-Binding instead of Branching:
  - Build/Package Options
  - Feature Configuration/Selection
  - Business Rules
- Think about which of the following needs to "vary" and what needs to stay the same:
  - Interface vs. Implementation vs. Integration
  - Container vs. Content vs. Context
- Commonality & variability analysis helps identify the core dimensions of variation for your project
- Use a combination of strategies based on the different types of needed variation and the "dimension" in which each one operates

96

Our Book



- Pub Nov 2002 By Addison-Wesley Professional

www.scmpatterns.com

97

Dr.Dobb's ARCHITECTURE & DESIGN WORLD DESIGN, MODEL AND BUILD SOLID FRAMEWORKS JULY 17-20, 2006, CHICAGO, IL

Other Agile SCM Resources

- http://www.scmpatterns.com/
  - *SCM Patterns* book has most of the codeline, workspace & build patterns presented here; and this site has a reference card and synopses for the patterns
- http://www.cmwiki.com/AgileSCMArticles
  - Numerous links to specific "Agile SCM" papers on the subject of patterns for continuous integration, promotion, staging, branching & merging, and more
- http://blog.bradapp.net/ and http://acme.bradapp.net/
- http://www.berczuk.com/
- http://www.cmcrossroads.com/

98

Thank You!


شكرا
Gracias
Juspajaraña
Teşekkürler
Salamat
Yum botic
Xie xie
Bedankt
Khawn khun
Obrigado!
Díky
ขอบคุณ
Arigato
Mahalo
谢谢
Merci
Vielen Dank
Spacibo
Danke
Grazie
Köszönettel
תודה

99