

# Common SCM Patterns in Software Development Projects

*Robert Ventimiglia, Project Manager; Matt Gelbwaks, Project Manager;  
Benjamin Kelleher, Sr. Consultant*

*Pretzel Logic Software, Inc.  
Suite 700, 210 Interstate North Parkway  
Atlanta, Georgia 30339  
(404) 980-6671  
[www.Pretzel.com](http://www.Pretzel.com)*

# Outline of Presentation

- *What are Patterns and why do I care?*
- *Software Development Project Description*
- *Common SCM Patterns that were unknowingly used when the project was setup*
- *Conclusions*

# SCM Pattern References

- *The following were used to identify the patterns unknowingly used on the project discussed and to prepare this presentation:*
  - *"Patterns in a Nutshell: The 'bare essentials' of Software Patterns", Brad Appleton, <http://www.enteract.com/~bradapp/docs/patterns-nutshell.html>*
  - *"Streamed Lines: Branching Patterns for Parallel Software Development", Brad Appleton, Stephen Berczuk, Ralph Cabrera, and Robert Orenstein; <http://www.enteract.com/~bradapp/acme/branching/>*
  - *"Software Reconstruction: Patterns for Reproducing Software Builds", Ralph Cabrera, Brad Appleton, and Stephen Berczuk; <http://www.enteract.com/~bradapp/acme/plop99/>*
  - *"Anti-Patterns and Patterns in Software Configuration Management", William J. Brown, et al, John Wiley & Sons; ISBN: 0471329290*

# Patterns Are:

- *A hot-topic, Object Oriented Design (OOD) buzzword*
- *Describe practical solutions to "real world" problems*
  - *Using a Structured file*
- *Larger solutions can be derived/achieved by using patterns as building blocks*
- *Possibly an analytical approach to creating Software Configuration Management (SCM) policies/procedures*

# Pattern Construction Format:

- ❏ **Name:** a meaningful “conceptual handle” for discussion
- ❏ **Context:** Describes how the problem/solution occurs including range of applicability, e.g. how you get/got there
- ❏ **Problem:** problem statement/solution
- ❏ **Forces:** constraints, goals, motivating factors and concerns
- ❏ **Solution:** solution structure, how to create the solution
- ❏ **Resulting Context:** Results obtained using the solution including pro/con tradeoffs used to determine correctness of the solution for a given culture/environment

# A Good pattern:

- ❏ *Solves a problem - captures solutions*
- ❏ *Is a proven concept - has a track record, not a theory!*
- ❏ *Solution isn't obvious - not derived from first principles, indirectly generates a solution to the problem*
- ❏ *Describes a relationship - deeper system structures and mechanisms*
- ❏ *Has significant human component - aesthetic and utility appeal*

- *An AntiPattern is a commonly occurring solution to a problem that has negative consequences*
  - *Is really a non-solution*
  - *It contains flaws*
  - *Causes more problems than the original problem being solved*
  - *Includes hints/tips for avoiding/prevention and recovery, e.g. a refactored solution that can be used to recover from the anti-pattern*
  - *Points to patterns that help begin the journey to recovery and renewal*

# Some Common SCM AntiPatterns

## ❏ *Silver Bullet AntiPattern*

❏ *The Lack of experience in SCM often leads to impractical solutions that don't work and eventually cause project failure. Most frequently is to rely on a tool to implement an SCM program*

## ❏ *SCM Expert AntiPattern*

❏ *SCM is not rocket science or quantum physics. SCM is conceptually simple: id and document what you want to control, control it, maintain accounting of changes, and check to make sure the documentation and product accurately reflect one another. As simple as it is the system engineer must be experienced enough to be able to apply and understand system engineering principles. Project experience, in the trenches with dirty finger nails is critical to being able to apply the SCM system engineering discipline.*

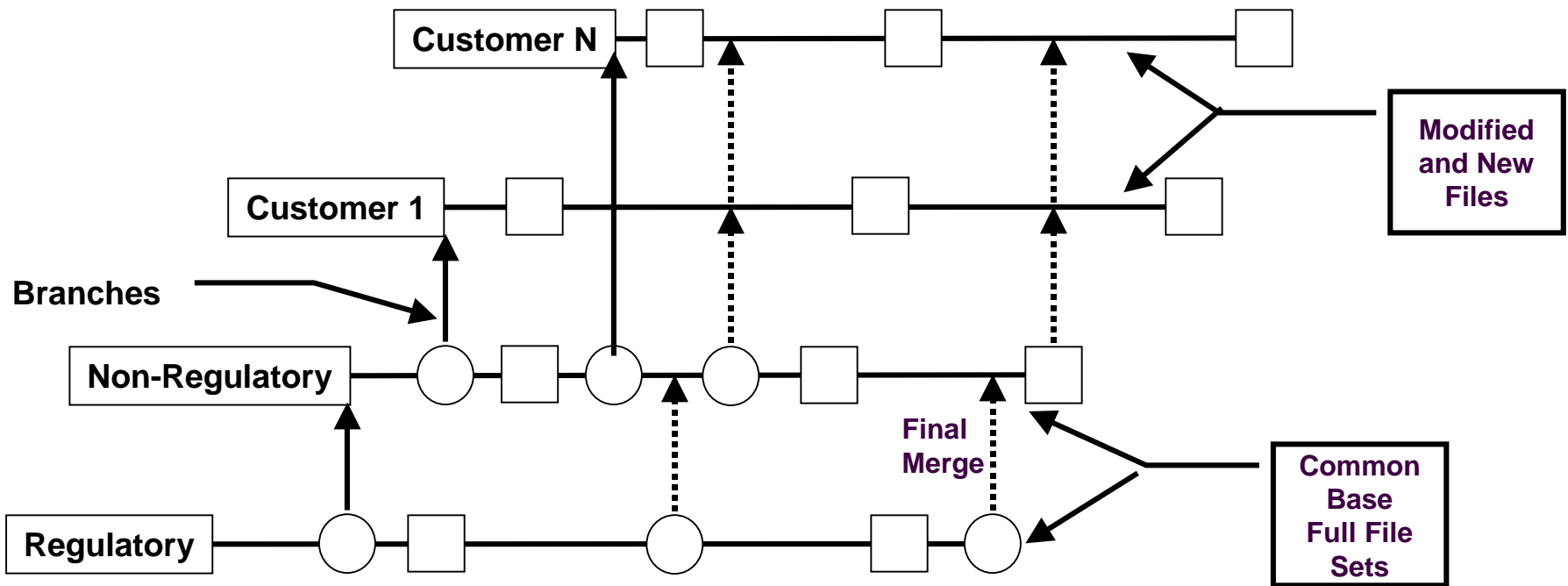
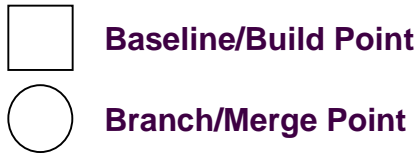
# Software Development Project Description

- *C130J Software Development*
- *8 Variations of 8 Software Configuration Items (SCI's)*
  - *Two Safety Critical SCI's*
  - *Common bases of reused software*
    - *Regulatory*
    - *Non-Regulatory*
  - *Simultaneous Flight Test and ongoing software development*
  - *Multiple versions of software in test*
    - *Problem reports being collected against all versions*
    - *Problem Correction coordinated/implemented in future versions*
  - *Embedded software development*
  - *Languages: Ada, C/C++, Rogue Wave on Unix*
  - *RTM, Interleaf, PVCS Dimensions, TeamWork, Rational/Apex Suite, Aonix Cross Compilers, Custom Interface Definition/Management, Custom Automated Test Environment*
- *> 150 developers*
- *24X7 development and flight test*

# Project Development & Build Architecture

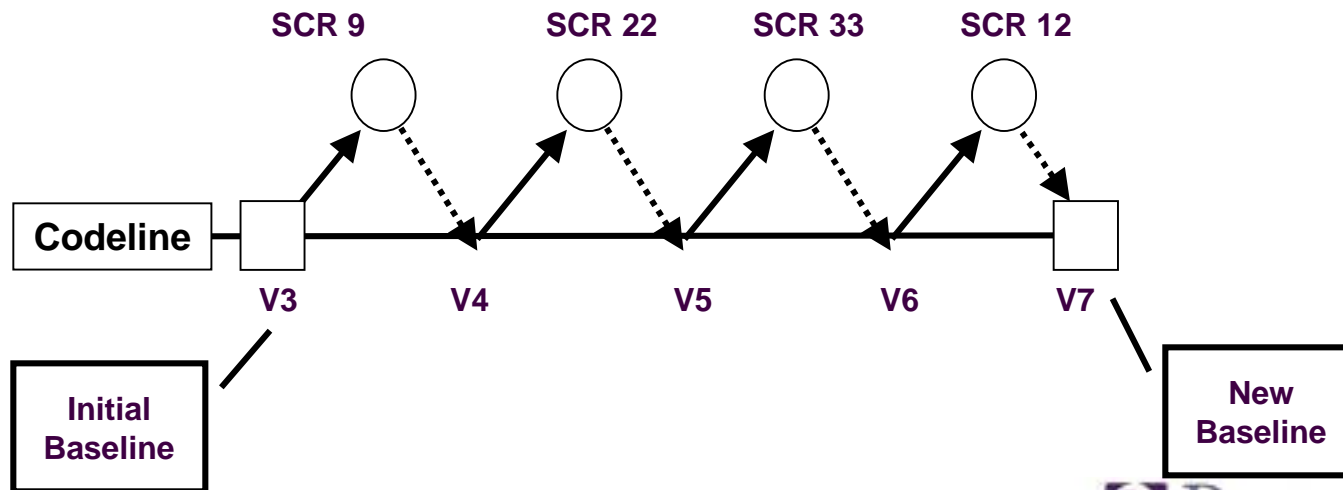
- ❑ *Process Based, Lifecycle Management tool used*
- ❑ *Single Primary Workspace per Base and Customer*
  - ❑ *Named Branch per Workspace*
  - ❑ *New Workspace and Named Branch for each variation*
  - ❑ *Lead Engineer per Workspace*
  - ❑ *Customer & Variant Workspaces contain only Modified and New files*
  - ❑ *Role based access*
- ❑ *Weekly Builds per Workspace (preferred)*

# Project Development Architecture



# Project Software Change Architecture

- ❑ *Software Change Request (SCR) required for all file check outs and updates*
  - ❑ *Electronically enforced by tool*
- ❑ *Single change per SCR*
  - ❑ *Sequential changes between baselines and builds*
- ❑ *Baseline creation driven by change requests*
  - ❑ *Created electronically using the tool*
- ❑ *Build Baseline constructed by merging Workspace Baseline onto Common code baseline*



# Common SCM Patterns Used

- *Mainline*
  - *Have all other codelines eventually join back into a single primary codeline*
- *Codeline Policy*
  - *Define a policy for each codeline which specifies if/when and how changes may be checked-out, checked-in, and merged, and propagated*
- *Codeline Ownership*
  - *Assign a responsible owner for each codeline to settle policy disputes and ensure the integrity and consistency of the codeline*
- *Merge Early and Often*
  - *Merge changes from a branch to its codeline as soon as the changes on the branch are completed and tested*
- *Restricted Access Line*
  - *When determining the Codeline Policy decide on the access-control policy of the Codeline*

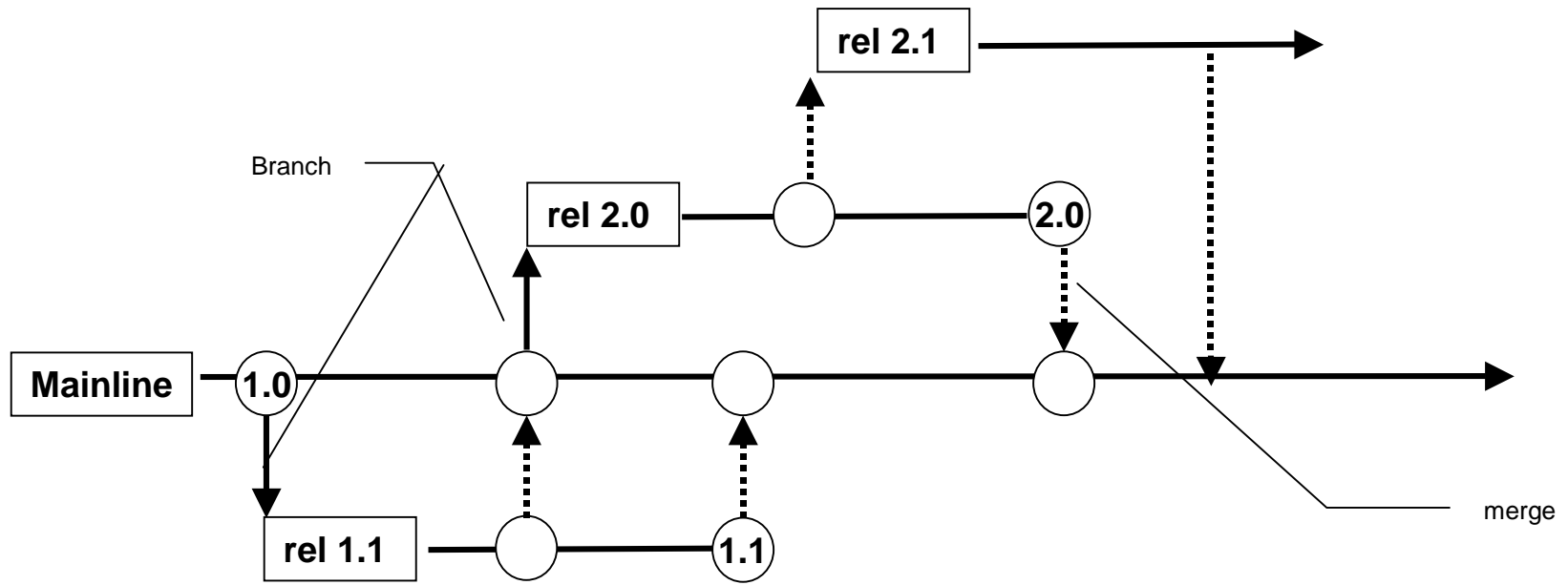
# Common SCM Patterns Used

- *Bill of Materials*
  - *You can successfully build the software system today and need to be able to build the same version in the future*
- *Reproducible Build*
  - *You've built the software system and created a build process, and need to know that it can be reproduced.*
- *Shared Object Cache*
  - *Allows developers to perform local builds based on extracted files and a common set of object files.*
- *Shared-Source Escalation*
  - *Maintain and support a common set of code used by more than one project.*



- *Solution:*
  - *Keep a home branch or trunk*
    - *avoids cascading/wide and unwieldy branches*
  - *Merge new releases back to the mainline branch - before creating a new branch*
    - *synching reduces change propagation across branches*
- *Resulting Context:*
  - *Reduces merging and synchronization effort*
    - *Requires fewer transitive change propagation's.*
  - *Keeps full revision names to a manageable length*
    - *Avoids maximum command line length limit*
  - *Provides closure (closing the loop)*
    - *Brings changes back to the "Mainline" of development*
    - *No splintering or fragmentation*
- *Project Usage:*
  - *Maintained two mainlines – regulatory and military*
  - *Customer Variants merged with mainline to produce customer buildable/testable baseline*

# Mainline Diagram



- ❏ *Aliases: Policy per Codeline*
- ❏ *Context: Development using multiple codelines*
- ❏ *Problem: Knowing which codeline to save to and when.*
- ❏ *Forces:*
  - ❏ *Different purpose per codeline*
  - ❏ *Codeline name is descriptive of its purpose/function*
    - ❏ *Does not express finer points of its usage*
  - ❏ *Saving to wrong codeline must be corrected, loses time*
  - ❏ *Detailed documentation of codeline policy takes time*
  - ❏ *Documentation may be perceived as over planning, non-value added*

## ☒ *Solution:*

- ☒ *Define naming conventions, meaningful branch names*
- ☒ *Be brief, specify only what is needed, one or two pages, few paragraphs*
  - ☒ *Function/purpose*
  - ☒ *Check-out/in, branch/merge, access control business rules*
  - ☒ *Interface with other codelines, import/export, propagation/reception*
  - ☒ *Expected duration, workload, frequency of integration, and retirement rules*

## ☒ *Results:*

- ☒ *Coherently communicated Codeline purpose*
  - ☒ *Available to the entire project team*
  - ☒ *Define how the codeline can and should be used.*
- ☒ *Codeline access issues resolved*
  - ☒ *Do they violate the intended purpose of the codeline?*
- ☒ *Documentation overhead/maintenance kept to a bare minimum*
  - ☒ *What's left is worth the effort/adds value*

- *Project Usage:*
  - *Each Customer's codeline was developed in separate workspaces using role based access control*
  - *Each workspace implemented named branches to simplify identification of changes made to mainline code.*
    - *Also simplified identification of which versions of a file belonged to which customer when looking at the file version tree*
  - *Change document electronically associated with specific versions of files to be modified*
    - *Helped ensure correct version of file was checked out*
  - *File Check out/check in required an approved change document*

- *Aliases: Branch Ownership*
- *Context:*
  - *Codeline policy does not define a needed activity or is vague*
- *Problem: Should the activity be performed? How to decide?*
- *Forces:*
  - *Policy cannot cover all situations, policy covers theory!*
  - *Developer needs policy clarified*
  - *Policy can be violated - accidentally or intentionally*
  - *Maintain codeline integrity and consistency by keeping it correct*

# Codeline Ownership, cont.

- *Solution: Assign one owner per codeline*
  - *Provide authoritative access control*
  - *Clarify whatever is unclear*
  - *Arbitrate/decide violations of policy*
  - *Assist in integration activities*
  
- *Resulting Context:*
  - *Single individual held accountable for codeline consistency/integrity*
    - *Codeline more likely to be in a consistent/reliable state.*
  - *Decreases the likelihood codeline policy being violated, or being used for the wrong purpose*
  - *Conceptual integrity maintained:*
    - *Single point of authority for resolving codeline issues;*
  
- *Project Usage:*
  - *Each workspace had a lead engineer assigned who decided who got access to the workspace and what roles they received*
  - *Lead made the work assignments*

# Merge Early and Often

- *Aliases: Codeline update per task*
- *Context:: Task branches need to be merged into mainline where development & maintenance is occurring*
- *Problem: When and how frequently should changes be merged?*
- *Forces:*
  - *Preserve integrity, traceability, consistency, and correctness*
  - *Frequent merges may upset stability, cause synchronization difficulties*
  - *merging may provide early surfacing of issues/risks*
  - *Can complicate developers efforts when multiple check-outs allowed*
- *Solution:*
  - *Integrate new changes in their entirety ASAP*

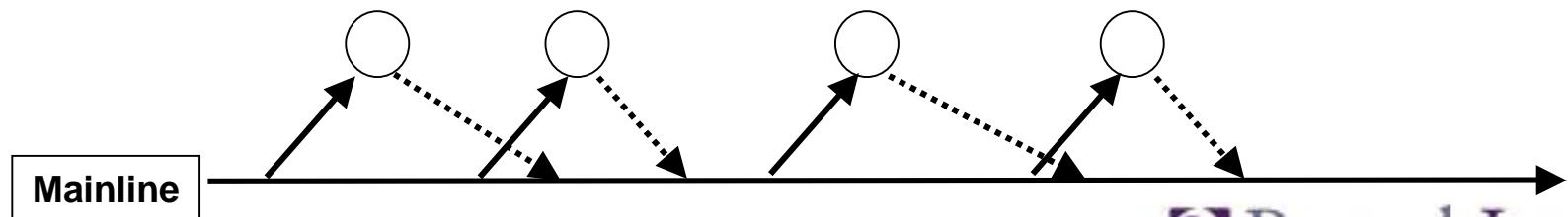
# Merge Early and Often, cont.

## Resulting Context:

- *Codeline incremental updates at regular intervals reflects current state of progress.*
  - *Subsequent development takes place in the context of newly merged changes.*
  - *May introduce difficulty if a change later needs to be backed out*
  - *Serves as a forcing function to identify/flesh out risks early in the development cycle*
- *Developers using dynamic configuration selection may see changes sooner than they would have liked. Although they would have to see them eventually before merging back to the codeline, they may require some control over when they see such changes that would impact their own work.*

## Project Usage:

- *Sequential changes made to files*
- *Weekly builds implemented to reduce integration issues*



- *Context:*
  - *When determining the Codeline Policy for a codeline, you need to decide the access-control policy of the codeline*
- *Problem:*
  - *How restrictive or exclusionary should the access control policy be for the codeline?*
- *Forces:*
  - *Tight controls may be necessary if there are inexperienced developers*
  - *Significant level of risk or complexity requires close monitoring and/or verification*
  - *Stability is important to avoid adversely impacting the development team*

# Restricted Access Line, cont

## ■ *Solution:*

- *Use an SCM tool that supports role based access to codeline workspaces.*
- *Or use an SCM tool that supports access control for branches so that they may be locked against check in or checkout.*
- *Or, If the SCM tool does not support such features, then scripts will need to be written for check in/checkout to a designated codeline.*

## ■ *Resulting Context:*

- *Restricted-access provides an added level of safety and security for the codeline. workspace*
- *Ensures that development in the codeline workspace remains as stable as it requires*

## ■ *Project Usage:*

- *Team used an integrated process management tool that supports role based access controls on access to workspaces.*
- *Development within the codeline workspaces remained stable while maximizing team productivity*

## Context:

- You can successfully build the software system today and need to be able to build the same version in the future

## Problem:

- How can you reproduce the build if you know that more than your source is required to build the software system?

## Forces:

- Components are not co-located on same system.
- System is complex and/or large
- Software build processes are complex
- Previous builds of software system must be reproduced

# Bill of Materials, Cont.

## ■ *Solution:*

- *Document all of the components that contributed to the build in a list in a file, i.e., a bill of materials (BOM). The BOM may contain the names, versions, and directory paths of operating systems, libraries, compilers, linkers, make-files, build scripts, etc. The BOM may be manually created, but many configuration management tools generate it as a by-product of the build.*
- *Place the BOM file under version control and associate it or include it in the baselines that it documents.*

## ■ *Resulting Context:*

- *The bill of materials identifies what components you need, where they can be found, what versions they are, and how to assemble them to reproduce the software system.*
- *The important purpose of the BOM is to identify components that are not under version control directly (e.g., environment information, compilers, linkers, et al).*

## ■ *Project Usage:*

- *A S/SEE Index was maintained that documented the environment versus time.*
- *Build scripts were developed, managed and baselined along with source code and test scripts*

## Context::

- ▣ *You have successfully built the software system once in the past and as a result, created a build process.*

## Problem:

- ▣ *How do you know if the build process and/or bill of materials can faithfully reproduce the software system?*

## Forces:

- ▣ *The build process may not capture all of the components (completeness).*
- ▣ *The build process may not identify the correct versions and locations of components (correctness).*

## ■ *Solution:*

- *Test both the build process and bill of materials by producing a build from them and checking for differences between the initial build and the process-generated build.*

## ■ *Resulting Context:*

- *Proven ability to repeat any build.*

## ■ *Project Usage:*

- *Tool created Baseline contained build script and being database driven was always re-executable.*
  - *Provided the underlying database and environment was properly maintained and backed up*
- *This was demonstrated repeatedly for regulatory audits.*

# Shared Object Cache

## Context:

- Developers perform local builds based on extracted files and a common set of object files. Unless other action is taken each developer will need a complete set of files extracted in order to perform local builds.

## Problem:

- How do you eliminate redundancy of effort with every developer compiling the same set of common file versions?

## Forces:

- The amount of disk space required on the developer's local platform to contain the object files produced by the entire source file set.
- The processing power and system overhead required to manage private workspaces for each developer.
- Repetitive Compilation time to recompile common files.

## ❏ *Solution:*

- ❏ *Maintain a pool of derived (compiled) objects with associated information. Developers' linkage paths point to their own local workspace first followed by the common pool.*

## ❏ *Resulting Context:*

- ❏ *Improved performance in total build time. Common file set is kept and managed in shared workspace, reducing the maintenance and build overhead for every developer*

## ❏ *Project Usage:*

- ❏ *Read-Only common object libraries for all in-use builds were maintained*
- ❏ *Developers were able to rapidly verify that changes compiled and passed unit tests.*
- ❏ *Improved team productivity and communications.*

# Shared-Source Escalation

## Context:

- You need to have similar products/projects underway. You recognize that there are components of these projects that will be identical or have common components that have not diverged.

## Problem:

- How can you support and maintain a common set of components used by more than one product/project?

## Forces:

- Portions of products/projects are similar.
- Product groups are providing a family of related products to provide flexible solutions to customers.
- Components or sub-products are discrete.
- There is overhead to supporting re-use.
- Product groups are challenged to bring products to market faster
- Maintaining two or more sets of identical components is difficult and prone to error.

# Shared-Source Escalation

## ■ *Solution:*

- *Make internal projects out of common code reused by multiple other projects.*
  - *identify the common components/products used by other projects,*
  - *establish an independent project for these common components*
  - *establish an internal merge/release process of the common components to the individual projects*

## ■ *Resulting Context:*

- *Shared components or products are maintained in one place*
- *A CCB (change-control board) can be put in place to coordinate and approve changes and enhancements of the common components.*
- *Over time different projects may find that some components need to be specialized. At that point they could absorb and take responsibility for their variant of the common components.*

## ■ *Project Usage:*

- *Common codeline and baseline was maintained*
- *Customer variant codelines merged onto common codeline to produce customer variant build-able code set.*

- ❑ *A number of common SCM Patterns were unknowingly utilized on the C130J Software Development Project*
- ❑ *New Patterns, yet to be completely described may have been uncovered;*
  - ❑ *Process Driven Development*
  - ❑ *Change Management Repository*
- ❑ *Some false starts could have been avoided if the Project had prior knowledge of these patterns*
- ❑ *SCM Patterns should be used to architect the implementation of at least complex development efforts*

**Patterns are Good!**

- *IPR: Integration Problem Report*
- *OOD: Object Oriented Design*
- *SCR: Software Change Request*
- *SCM: Software Configuration Management*