Streamed Lines: Branching Patterns for Parallel Software Development

Brad Appleton	*bradapp@enteract.com*	Motorola Cellular Infrastructure Group
Stephen P. Berczuk	berczuk@acm.org	Biztravel.com
Ralph Cabrera	cabrerar@agcs.com	AG Communication Systems
Robert Orenstein	rlo@perforce.com	Perforce Software

Abstract: Most software version control systems provide mechanisms for branching into multiple lines of development and merging source code from one development line into another. However, the techniques, policies and guidelines for using these mechanisms are often misapplied or not fully understood. This is unfortunate, since the use or misuse of branching and merging can make or break a parallel software development project. *Streamed Lines* is a pattern language for organizing related lines of development into appropriately diverging and converging streams of source code changes.

Keywords: *Branching, Parallel Development, Patterns, Software Configuration Management, Version Control*

Introduction to SCM Patterns

The approach an organization takes to Software Configuration Management (SCM) can significantly affect the quality and timeliness with which a software product is developed. By SCM, we essentially mean the process of identifying, organizing, controlling, and tracking both the decomposition and re-composition of software system structure, functionality, evolution, and teamwork. In short, SCM is the "glue" between software components, features, changes, and team members. This paper presents some of the patterns from a pattern language for SCM that we began developing at ChiliPLoP '98.

Motivation for an SCM Pattern Language

There are many approaches to SCM, and the structures, policies, and processes work best when applied in the appropriate context. Context is determined by organizational and architectural decisions, as well as previously existing SCM policies. Our goal is to place SCM structures in the context of these other existing structures, making it easier to decide how to structure an SCM process which will be effective for your context.

These SCM structures may be described as "*patterns*": named nuggets of insight conveying battle-proven solutions to recurring problems, each of which balances a set of competing concerns (see [Appleton97]). SCM Patterns fit into a framework of *Organizational Patterns*, which can be grouped as follows:

Organizational Patterns:

Patterns which define how the organization is structured. This includes patterns that describe the size of the team, management style, etc. (see [Beedle97] and [OrgPats]).

Architectural Patterns:

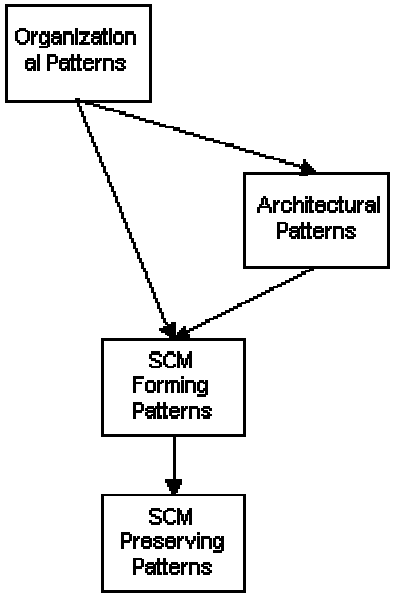
Patterns which define how the software is structured at a high level. Some examples of these sorts of patterns have been published in prior PLoP Proceedings ([Berczuk95] and [Berczuk96]) and in books such as [POSA].

Process Defining (forming) Patterns:

These SCM Patterns describe structures, such as the project directory hierarchy, which are set up near the beginning of a project.

Maintaining (preserving) Patterns:

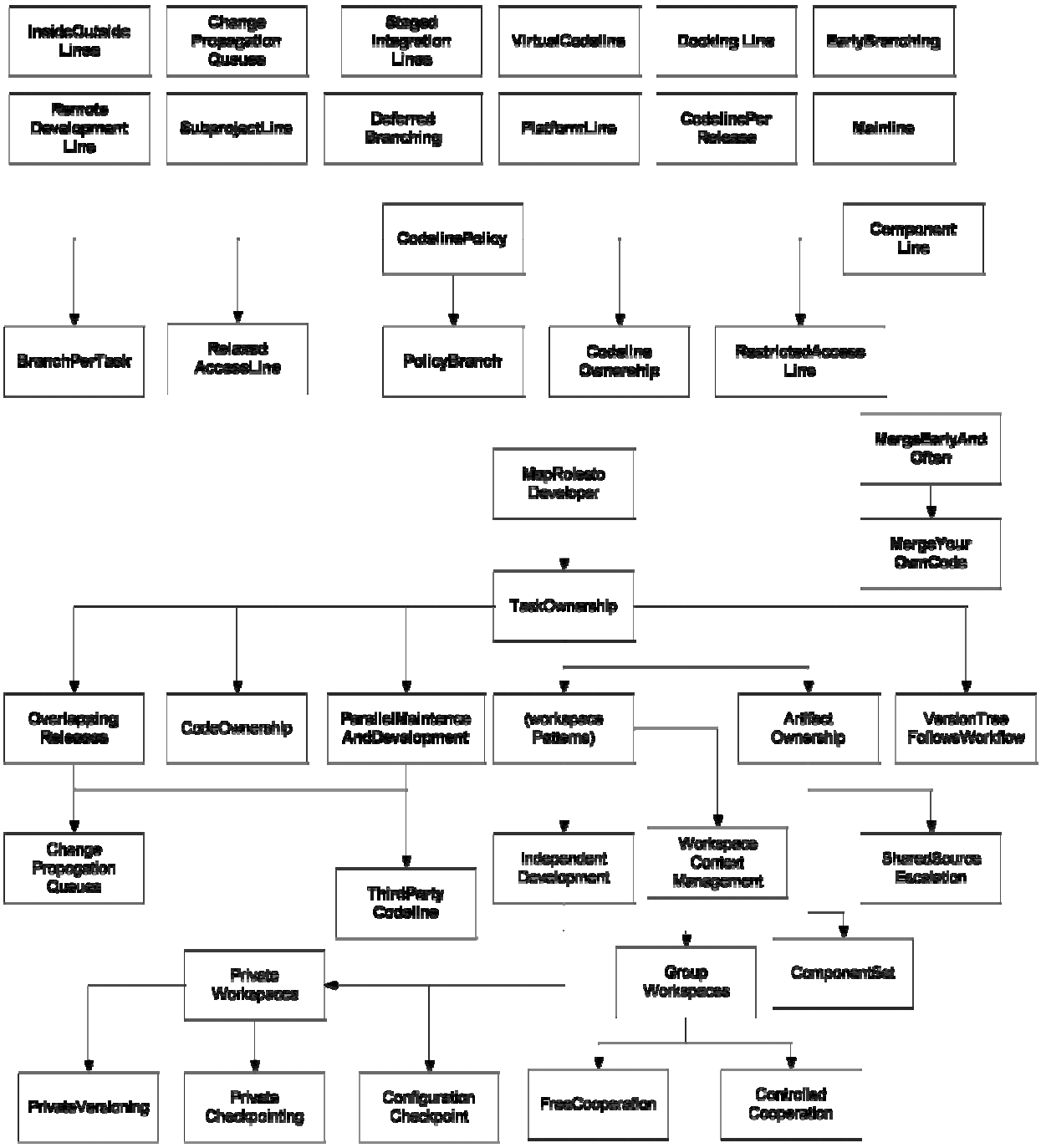
These are SCM patterns that affect the day to day workings of the organization.



These categories of patterns are shown in the figure at right.

The line between the Forming and Maintaining patterns may be blurry, but we feel the distinction is conceptually important to understand the architecture of the development process pattern language. Because of the strong relationship between the patterns in each category (how you set up the directory tree affects the process you follow for checking files in and out) we shouldn't spend too much time looking at where a pattern fits, but rather focus on which patterns it follows from.

The patterns we are documenting, including the ones we present here, should be applied with an understanding of the context in which the problem exists. The figure below shows the patterns we are working on, with their relationships in the language are in bold face. This paper provides an in depth discussion of some branching patterns for parallel development (their names are in bold face in the figure below). Some of the other kinds of patterns have been published previously ([Berczuk97]). Others are in "patlet" form now.


InsideOutside Lines
Change Propagation Queues
Staged Integration Lines
VirtualCodeline
Docking Line
EarlyBranching
Remote Development Line
SubprojectLine
Deferred Branching
PlatformLine
CodelinePer Release
Mainline
CodelinePolicy
Component Line
BranchPerTask
Relased AccessLine
PolicyBranch
Codeline Ownership
RestrictedAccess Line
MergeEarlyAnd Often
MapRoleto Developer
MergeYour OwnCode
TaskOwnership
Overlapping Releases
CodeOwnership
ParallelMaintenance AndDevelopment
(workspace Patterns)
Artifact Ownership
VersionTree FollowsWorkflow
Change Propogation Queues
ThirdParty Codeline
Independent Development
Workspace Context Management
SharedSource Escalation
Private Workspace
Group Workspaces
ComponentSet
PrivateVersioning
Private Checkpointing
Configuration Checkpoint
FreeCooperation
Controlled Cooperation

Parallel Development

Any software project of certain team and system sizes will inevitably require at least some efforts to be conducted in parallel. Large projects require many roles to be filled: developers, architects, build managers, quality assurance personnel, and other participants all make contributions. Multiple releases must be maintained, and many platforms may be supported. It is often claimed that *parallel development* will boost team productivity and coordination, but these are not the only reasons for developing in parallel. As [Perry98] points out, parallel development is inevitable in projects with more than one developer. The question is not "should we conduct a parallel development effort", but "*how* should a parallel development effort best be conducted?"

[Perry98] suggests that many of the basic parallel development problems that arise can be traced back to the essential problems of: system evolution, scale, multiple dimensionality, and knowledge distribution:

- Evolution compounds the problem of parallel development because we not only have parallel development within each release, but among releases as well.
- Scale compounds the problem by increasing the degree of parallel development and hence increasing both the interactions and interdependencies among developers.
- Multiple dimensions of system organization compound the problems by preventing tidy separations of development into independent work units.
- Distribution of knowledge compounds the problem by decreasing the degree of awareness in that dimension of knowledge that is distributed.

Thus, a fundamental and important problem in building and evolving complex large-scale software systems is how to manage the phenomena of parallel changes. How do we support the people doing these parallel changes by organizational structures, by project management, by process, and by technology?

Streamed Lines: Branching for Effective Parallel Development

If parallel development is a fact of life for any large software project, then how can developers making changes to the system in parallel be supported by project management, organizational structures, and technology? *Streamed Lines* is a pattern language that attempts to provide at least a partial answer to this question by presenting branching and merging patterns for decomposing a project's workflow into separate lines of development, and then later recomposing the development lines back into the main workstream. The patterns describe recurring solutions for deciding how and when development paths should diverge (branch) and converge (merge). *Streamed Lines* does not describe a complete solution to all the problems encountered during parallel development; in fact, these patterns are merely a branching sublanguage of a larger SCM pattern language currently in development.

Effective Parallel Development

What do we even mean by "effective parallel development"? [Atria95] defines *effective parallel development* as:

> ... the ability for a software team to undertake multiple, related development activities -- designing, coding, building, merging, releasing, porting, testing, bug-fixing, documenting, etc. -- at the same time, often for multiple releases that use a common software base, with accuracy and control.
>
> Note that this definition extends to include teams that span multiple locations, an increasingly common situation for many organizations. It encompasses all elements of a software system and all phases of the development lifecycle. Inherent within the definition is the concept of integration, in which parallel development activities and projects merge back into the common software base. Also, the definition of effective parallel development includes process control -- the policies and "rules of the road" that help assure a controlled, accurate development environment.

So how can we achieve effective parallel development? While it is by no means a complete solution, one way that helps is the effective use of branching.

Introduction to Branching

The following is a brief introduction to the concepts of file checkin/checkout, and to branching and merging. If you are already familiar with these concepts you may safely skip this section.

The checkin/checkout model

Most version control tools in widespread use employ the *checkout-edit-checkin* model to manage the evolution of version-controlled files in a repository or *codebase*. Developer's checkout a file from the codebase into their workspace where they perform and then test their modifications. When they are finished, they checkin their modified source file revisions back into the codebase. No files may be modified unless they are first checked-out.

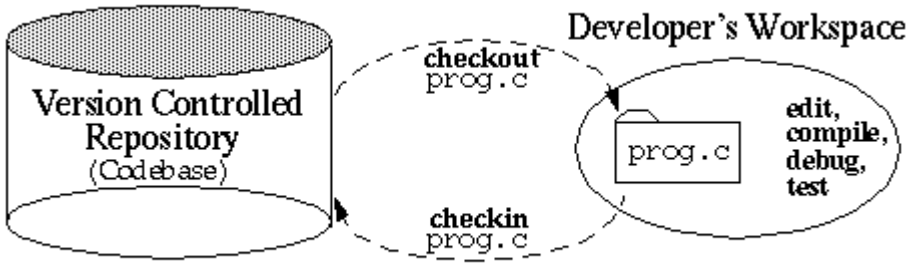


Figure 1: The checkout-edit-checkin model of version control

Serial development using exclusive checkouts

In a strictly sequential development model, when a developer checks-out a file, the file is write-locked: No one may checkout the file if another developer has it checked-out. Instead, they must wait for the file to be checked-in (which releases or removes the write-lock). This is considered a pessimistic concurrency scheme which forces all work to take place on a single line of development.

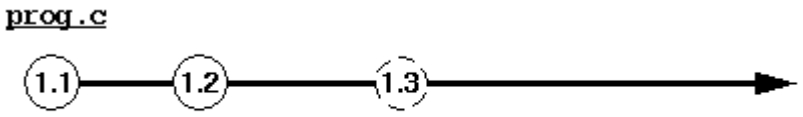


Figure 2: Serial development line for file `prog.c`

Concurrent development using branches

Branching is a common mechanism used by many version control (VC) tools to support concurrent software development. In its most basic form, branching allows development to take place along more than one path for a particular file or directory. If one developer wants to checkout a file that another developer has already checked-out, she merely creates a branch and then checks-out the file on the new branch. This is an optimistic concurrency scheme that creates a parallel line of development for the file.

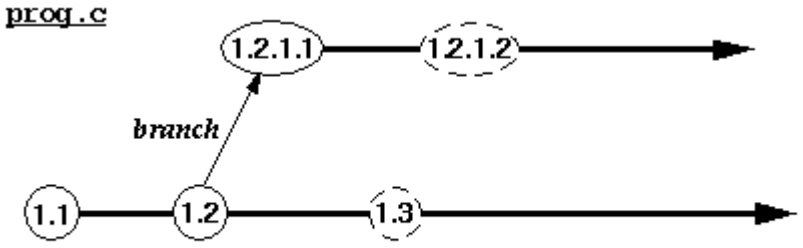


Figure 3: Branching off a new development line for file `prog.c`

Branching can be likened to a Unix `fork` that creates a new thread of execution with copy-on-write semantics. When a branch is created, the contents of the revision serving as the *branchpoint* are the same as the contents of the

initial revision on the newly created branch. When the revision on the new branch is modified and checked-in, the two lines of development will have different contents and will evolve separately, in isolation from one another.

Synchronizing concurrent development lines using merges
Merging is the means by which one development line synchronizes its contents with another development line. When merging from a child branch back to a parent branch, the contents of the latest version on the child branch are reconciled against the contents of the latest version on the parent branch (usually using a 2-way or 3-way file differencing or comparison tool). The contents of the two files are merged together into a new revision which must correctly resolve any conflicting changes made along each development line since the branchpoint was created (or since the last time the two development lines were synchronized).

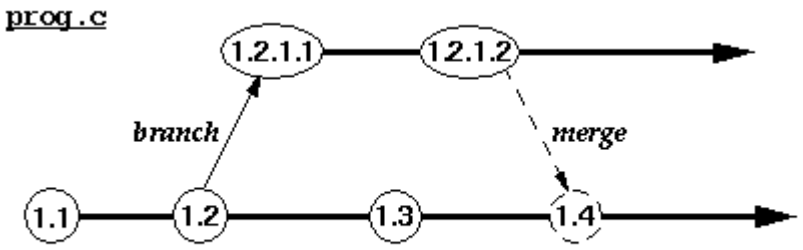


Figure 4: Merging back to the parent branch for file `prog.c`

Dimensions of Branching

So far we have discussed checkout/checkin (and branching) of files and directories. Files and directories are *physical configuration elements* for software. Most VC tools that support branching let you checkout/checkin files, and perhaps directories. Some VC tools are integrated with programming environments and let you checkout/checkin programming language elements like classes and methods (or modules and functions). The discussion that follows assumes that files and directories are the minimal configuration elements, but the ideas and strategies presented here are applicable to VC tools that allow branching of methods and classes.

Most VC tools supporting branches do so at the granularity of a lone file or element. The revisions and branches for each file form a *version tree* that depicts the evolution of a single file. But branching is most conceptually powerful when viewed from a project-wide or system-wide level, where the resultant version tree reflects the evolution of an entire project or system. There are essentially five different forms of branching, each of which may be represented using the file-based branching of most VC tools:

Physical:
Branching of the system's physical configuration - branches are created for files, components, and subsystems

Functional:
Branching of the system's functional configuration - branches are created for features, logical changes (bug-fixes and enhancements), and other significant units of deliverable functionality (e.g., patches, releases, and products)

Environmental:
Branching of the system's operating environment - branches are created for various aspects of the build and run-time platforms (e.g. compilers, windowing systems, libraries, hardware, operating systems, etc.) and/or for the entire platform

Organizational:
Branching of the team's work efforts - branches are created for activities/tasks, subprojects, roles, and groups

Procedural:
Branching of the team's work behaviors - branches are created to support various policies, processes, and states

Specific instances of each type of branching will be discussed in many of the patterns that follow. It should be

mentioned that there is frequent overlap between the above types of branching. For example, a branch created for a particular bug-fix may be regarded as both a bug-fix branch, and as an activity-branch. In this case, the sets of changes that constitute the fix are performed as a single task. But a branch created for an integration effort won't always correspond to a single fix or feature. It is quite common, however, for a branch to correspond to more than type of branching. The important thing to remember is which type is perceived as the primary intent of the branch.

Branching Terms and Notation

Branches, Change-Tasks, and Codelines
In general, when a branch corresponds to a line of development containing (or intended for) multiple sets of logical changes, we refer to the branch as a *codeline*. Often, a branch is used only for a single logical change (also called a *change-task*). If a branch is used for a single change-task and is then immediately merged back to its parent, we call it an *activity-branch*, or simply a *branch* or *subbranch*. In theory, the terms "branch" and "codeline" may be used as synonyms. When describing branching patterns, however, we try to be consistent in using the term "codeline" to refer to a persistent workstream, and using the term "branch" to mean a subbranch of a codeline or a single activity-branch.

Versions, Change-Packages, and Baselevels
A *version* may refer to a revision of a single file, or to a set of files revision that make up the entire project (or one of its components/subsystems). A *change-package* is a configuration of the revisions that were modified or created as part of a change-task. A *baselevel* is a configuration of the project that is self-consistent enough to serve as a *stable base* for subsequent development efforts. A *baseline* is a baselevel that is suitable for an formal internal or external release.

Merging, Propagating, and Syncing
Merging is the process of integrating the revisions in a change-package into a the contents of a codeline. Sometimes, a change in one codeline needs to be incorporated into another codeline. For example, a bug-fix in a maintenance codeline may also needed in the corresponding development codeline for the next major release. We refer to this as *change propagation*, or simply *propagation*. When the entire contents of a codeline are merged into another codeline, or into a developer's workspace, we call this *syncing with the codeline*, or just *syncing*.

Version Tree Diagrams
Since revision names like "`1.4.1.2`", used by VC tools like RCS (and many others) aren't particularly mnemonic, we use more symbolic branch names consisting of letters and numbers (and some other characters). We also use the '`/`' character to indicate the beginning of a branch name, so that versions can be uniquely determined with an identifier such as "`/main/rel1-maint/fix232/4`". Hence a fully specified version name resembles a directory path in Unix or DOS. A few VC tools (most notably ClearCase and Perforce) use the same or similar conventions for version naming.

When drawing codelines, branches, change-tasks, and their relationships, we use a tree structure with branch-names inside boxes and version-names inside circles (a "box" or "circle" with no name inside is considered "anonymous"). Branches and codelines are indicated with solid lines, whereas merges and propagations are indicated with dashed lines. These *version-tree diagrams* are reminiscent of interaction sequence diagrams in the UML; but we draw the timeline from left to right instead of from top to bottom (to conserve space). Branch names always appear at the beginning of the timeline for the branch, and are preceded by a '`/`'. When a "box" appears in the middle of a timeline for a branch, it corresponds to a change-task that was made "on-line" (directly on the codeline, instead of on its own branch), and there is *no* leading slash in front of the name for such a change-task.

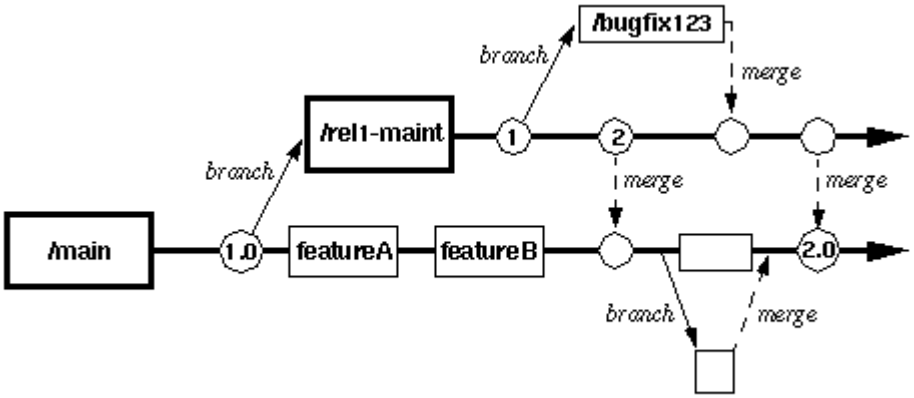


Figure 5: A sample version tree diagram for a project or system

Forces of Branching and Parallel Development

Safety
Safety is the property that nothing bad ever happens to your project. Safety concerns for parallel development include:

- *Codeline Consistency*: You want to keep the codeline in a consistent state. That means it has all the changes it is supposed to have (no partial changes), and that they have been successfully integrated so they don't conflict with one another.
- *Codeline Reliability*: Other developers rely upon the current state of the codeline to build and function correctly. If it doesn't, work may be held-up and efforts may be wasted.
- *Codeline Integrity and Stability*: At any given point in time, you want the codeline to be both consistent, and reliable. Over any given period of time, you want to codeline to be both reliably consistent, and consistently reliable. Otherwise changes which work correctly in isolation might fail when combined together. A single unanticipated change could destabilize the entire project.
- *Lost Changes*: Parallel development without controlled interaction between concurrent changes can result in lost or corrupted changes, and wasted effort and rework.
- *Reappearing Bugs*: Strategies need to be employed to ensure that defects in a prior version don't reappear in a later version of the product (especially when multiple versions are being developed concurrently)

Liveness
Liveness is the property that progress on the project ever takes place. Liveness concerns for parallel development include:

- *Increased Work Efficiency/Productivity*: Increasing concurrency in a development project can permit more work to happen at once, resulting in decreased time-to-market.
- *Increased Coordination Efforts*: Increasing the amount of concurrency increases the need for, and amount of, synchronization efforts (merging and integration). Merging and integration are often non-trivial activities that can be both time consuming and error prone. If taken too far or not planned intelligently, the synchronization overhead required by locking and merging can outweigh any productivity gains due to parallelizing development.
- *Contention and Work Stoppages*: Excessive locking and "work queuing" may cause very long "busy waits" where people are forced to suspend some of their activities until a locked file becomes available. This could result in dormancy or even deadlock of parallel activities.

Reusability
Reusability is the property that baselevels and changes can be readily used and incorporated in derived configurations while still functioning correctly. Reusability concerns for parallel development include:

- *Reproducibility*: If you can't reproduce the contents of a change or a baselevel, then you can't easily reuse it in developer's workspaces or in derived configurations.
- *Traceability*: If you can't readily find and identify the contents of a change or a baselevel, then you can't easily reuse it.
- *Separability*: If you can't readily disentangle desired changes from unwanted changes or baselevels, then you can't easily reuse those changes.

Teamwork

Communication, effort, and interaction need to be effectively organized and executed for a parallel development effort to succeed. Important issues are:

- *Roles and Responsibility*: If everyone is responsible for a thing, then often no one is. Features, changes, components, and milestones need responsible owners who understand their purpose and are held accountable for the success of their outcomes.
- *Coordination*: Team members need to stay reasonably abreast of what other team members are doing and why. But parallel efforts need to be insulated from one another and coordinated together so that changes can be successfully isolated and integrated. At the same time, you need to be careful that you don't go to far and isolate team members their own efforts, nor from each other.
- *Complexity*: The inherent complexity of changes and tasks needs to be managed. Complex tasks and changes need to be broken down into more manageable chunks and assigned to appropriate owners. Dependencies between tasks and between changes need to be minimized to reduce the risks associated with safety, liveness, and reusability mentioned above.

SCM Tool Support

Support mechanisms provided, or lacking in your SCM tools for change control/tracking and version control have a huge impact upon the success or failure of parallel development efforts. Some of these are as follows:

- *Branching Support*: Not all VC tools provide a conceptually easy way of employing branches and using them to group functional changes. Many VC tools (such as RCS [Tichy85] and SCCS [Rochkind75]) indicate branches using a hierarchical version numbering scheme such as `1.2.3.4`, where each subsequent component identifies a branch of its predecessor. Other VC tools (such as ClearCase [Leblang94], and Perforce [Seiwald96]) provide more symbolic hierarchical names, like `/main/rel_1.1/fix_file_menu`, which more easily lend themselves to comprehension by humans. CVS (see [CVS]) allows mnemonic labels called "sticky tags" to be used for branch names, but branch names are not hierarchically composed into branch paths.
- *Change-Packages*: Some SCM tools directly support the notion of logical changes (sometimes called *change-tasks* or *change-packages*): They are able to treat modifications to multiple files for the same coherent purpose as a single, logical change. This can be extremely useful for managing the functional configuration of a software product, and for identifying, tracking, and integrating modifications for a particular fix or feature.
- *Customization*: Some SCM tools are more configurable than others. As nice as a tool might be, it can't possibly provide everything you could want at the invocation of a single command, macro, or mouse-click. Some tools provide customization and extension primitives using a programming and/or GUI interface.
- *Extensibility*: Some tools provide "*triggers*": user-definable "hooks" or "plug-ins" that execute at a specific time during an SCM tool event or operation. Triggers can help provide a seamless integration between your tools and processes. If your tool doesn't support triggers or user-defined extensions for a desired function or feature, you may have to resort to implementing "*wrappers*": home grown scripts or commands that serve as a front-end to certain tool features, or which compose those features together in a useful manner.

Streamed Lines: Participants

The patterns in *Streamed Lines* are divided into categories of policy, creation, and structuring. These categories loosely correspond to the [GoF] pattern categories: behavioral, creational, and structural (respectively). In addition, many of the patterns refer to some basic types of branches and codelines. We define all of these categories below:

Basic Branch/Line Elements
Some basic varieties of branches and codelines that serve as lower-level building blocks for various patterns; these are not necessarily patterns per se, but they nevertheless participate in one or more patterns in the language

Branching Policy Patterns
Patterns describing behavioral policies to establish or preserve the conceptual or physical characteristics of a codeline

Branch Creation Patterns
Patterns describing when to create a new kind of branch or codeline

Branch Structuring Patterns
Patterns describing the collaborations between two or more related branches in a branching structure

The participants in *Streamed Lines* are distributed among these four categories as follows:

Basic Branch/Line Elements	**Branching Policy Patterns**
• E1. Activity Branch • E2. Functional Branch • E3. Component Branch • E4. Project Branch • E5. Development Line • E6. Maintenance Line • E7. Integration Line • E8. Release Line	• P1. Codeline Policy • P2. Codeline Ownership • P3. Relaxed-Access Line • P4. Restricted-Access Line • P5. Merge Early and Often • P6. MYOC (Merge Your Own Code) • P7. Early Branching • P8. Deferred Branching
Branch Creation Patterns	**Branch Structuring Patterns**
• C1. Policy Branch • C2. Branch per Task • C3. Codeline per Release • C4. Subproject Line • C5. Virtual Codeline • C6. Remote Line • C7. Component Line • C8. Platform Line	• S1. Mainline • S2. Parallel Maintenance/Development • S3. Overlapping Releases • S4. Docking Line • S5. Staged Integration Lines • S6. Change Propagation Queues • S7. Third Party Codeline • S8. Inside/Outside Lines

Extremely brief overviews for each basic element, pattern, and pattern variant are provided below. The full pattern descriptions appear in Appendix A.

Basic Branch/Line Elements

E1. *Activity Branch*
A branch used to represent a logically atomic, discrete unit of effort. The effort might be for effecting a fix or feature, or for building and releasing. If the emphasis is on the functional changes themselves rather than on the effort involved, we would call it a *Functional Branch* (see below). Related patterns are *Branch per Task*, *Merge Early and Often*, *MYOC (Merge Your Own Code)*, *Docking Line*, and *Change Propagation Queues*.

E2. *Functional Branch*
A branch used to represent discrete unit of logical functionality. If it is for a feature or an enhancement, we call it a *Feature Branch* (E2.1); if it is for a bug-fix, we call it a *Fix Branch* or a *Bugfix Branch* (E2.2). Since functional units often map to work-units, functional branch and activity branch are the same thing in most cases, but not always. A build, or a system integration might be a logical unit of work, but typically not of functionality.

E3. *Component Branch*
A branch or codeline that is primarily intended for work on a specific component of the system. It might be a lone module, *component*, subsystem, or an entire product within a multi-product project. The patterns *Component Line* and *Staged Integration Lines* make use of component branches.

E4. *Project Branch*
A branch or codeline that represents a separate line of development for a particular (sub)project consisting of more than *one* task or activity. It is essentially an activity-branch for a task consisting of multiple discrete activities (or subtasks). Related patterns are *Codeline per Release*, and *Subproject Line*.

E5. *Development Line*
A codeline upon which development (changes) takes place and which may or may not include maintenance efforts. In the broad sense, it includes maintenance; in the narrow sense, it is only for "new" development. Related patterns are *Mainline*, *Parallel Maintenance/Development*, and *Remote Line*.

E6. *Maintenance Line*
A codeline intended primarily for maintenance efforts (bug-fixes and minor enhancements), and possibly for release-engineering efforts as well. Related patterns *are Parallel Maintenance/Development*, and *Deferred Branching*.

E7. *Integration Line*
A codeline that is primarily for the purpose of integration (merges) from other codelines and branches. If no other type of work takes place on the integration line, we call it a *Receiving Line* (E7.1), because all it does is receive changes from other codelines. If, in addition to receiving changes from other codelines, it also propagates those integrated changes to other codelines, then we call it a *Shipping/Receiving Line* (E7.2). Related patterns are *Mainline*, *Docking Line*, *Staged Integration Lines*, *Merge Early and Often, MYOC (Merge Your Own Code)*, and *Change Propagation Queues*.

E8. *Release Line*
A codeline that represents a logical grouping of delivered functionality (a patch or a release). If it is known to be for a patch (as opposed to a release) we may call it a *Patch Line* (E8.1). If the codeline corresponds to all releases and patches for the same major release (e.g. all releases with the same major release number, as in "`1.x.y`"), then we may call it a *Major Release Line* (E8.2), but we will typically just use the blanket term "release-line" to refer to both types. Often, a release-line maps directly to a project-branch, but not always (this is similar to the relationship between functional-branches and activity-branches). Related patterns are *Parallel Maintenance/Development*, *Overlapping Releases*, *Codeline per Release*, and *Early Branching*.

Branching Policy Patterns

P1. *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.

P2. *Codeline Ownership*

Assign an responsible owner for each codeline to settle policy disputes and ensure the integrity and consistency of the codeline.

P2.1 *Codeline Dictatorship*

No one may perform any checkins, checkouts, or branches off a codeline without approval from the codeline-owner; or else no one at a remote site may do anything to a codeline at the master-site, without approval from the master-site.

P3. *Relaxed-Access Line*

Permit any developer who might have a need to perform checkouts, checkins, or branches for a codeline.

P3.1 *Guarded Merge Access*

Let developers checkin to the codeline as they see fit, contingent upon an automated check ensuring there will be no merge conflicts upon checkin.

P4. *Restricted-Access Line*

Apply a "lock" to a branch or codeline, or otherwise limit the set of users that may make changes to it to a small set of trusted developers.

P4.1 *Codeline Freeze/Retirement*

Lock (freeze) a codeline against all further development while it is being prepared for a release. Then, either decommission the codeline (leaving it locked), or else unlock the codeline after release efforts have completed to resume development.

P4.2 *Export Lock*

Briefly lock a codeline against all other checkins while a particular developer is merging a change into the codeline.

P4.3 *Private Branch*

Lock a codeline or branch against use by all but the codeline owner.

P4.4 *Role-based Line Access*

Rather than restricting a codeline to certain users, restrict it to certain development roles.

P5. *Merge Early and Often*

Merge changes from a branch to its codeline as soon as the changes on the branch are completed and tested.

P5.1 *Propagate Early and Often*

Propagate merged changes to target codelines as soon as the original change has been successfully integrated into its original codeline.

P5.2 *Multi-Merge Early and Often*

Rather than merging a single change-task at a time, merge a handful of change-tasks at once and do it at regular intervals.

P6. *MYOC (Merge Your Own Code)*

Developers are responsible for merging their own code into the codeline.

P6.1 *PYOC (Propagate Your Own Code)*

Developers are responsible for propagating their own code into other codelines.

P7. *Early Branching*

Spawn a new release or development codeline as soon as work for the new release or subproject begins.

P8. *Deferred Branching*

Wait to spawn a new release or development codeline until it starts to conflict with parallel work for its successor.

Branch Creation Patterns

C1. *Policy Branch*

Create a new branch when some users of an existing codeline need to work according to a different policy from the rest of the codeline's developers.

C2. *Branch per Task*

Perform each change-task on its own activity-branch.

C2.1 *Branch per Major Task*

Create an activity-branch only for sufficiently long or complex/risky activities. Perform minor tasks directly on the codeline.

C2.2 *Branch per Change-Request*

Perform each change-task resulting from a formal change-request on its own activity-branch.

C2.3 *Personal Activity Branch*

Create an activity branch for use by one and only one developer.

C3. *Codeline per Codeline*

Use a separate codeline for each major and minor release (and patches).

C3.1 *Codeline per Major Release*

Use a separate codeline for each major release.

C4. *Subproject Line*

Rather than having change-tasks depend on activity branches that haven't been merged into the codeline, create a subproject codeline for the related set of dependent tasks.

C4.1 *Personal Codeline*

Create a new codeline for related changes that will be performed by a single developer.

C4.2 *Experimental Codeline*

Create a new codeline for experimental development/prototyping, which might not get merged back into anything.

C4.3 *Multi-Project Lines*

Multiple project-lines or product-lines are used and releases are configured by selecting the appropriate stable baselevel of each project/product-line and integrating the result.

C5. *Virtual Codeline*

Simulate a codeline using a floating label, and just-in-time element branching.

C6. *Remote Development Line*

Create a new codeline specifically for development and integration to be performed at a remote-site.

C7. *Component Line*

Create a new codeline reserved exclusively for development and maintenance of a component-owner's source code.

C7.1 *Multi-Product Lines*

Same as *Multi-Project Lines* configured by selecting the appropriate stable baselevel of each project/product-line and integrating the result.

C8. *Platform Line*

A permanent development codeline for an operational variant of the system that must build/run on a different platform.

Branch Structuring Patterns

S1. *Mainline*
Have all other codelines eventually join back into a single primary codeline.

S1.1 *Stable Receiving Line*
Reserve the primary codeline exclusively for integration of stable bases from other codelines.

S1.2 *LAG Line*
Reserve a primary codeline for the latest and greatest development efforts.

S2. *Parallel Maintenance/Development Lines*
When line maintenance and development must occur simultaneously, split maintenance of into a separate codeline that regularly merges back into the development line.

S2.1 *Parallel Releasing/Development Lines*
Instead of freezing the codeline during release engineering activities, a separate line is created for release integration and engineering while allowing other development to continue taking place on the development line.

S3. *Overlapping Releases*
When developing for multiple releases at the same time, split the earlier release into a separate codeline that regularly merges back into the other release development line.

S4. *Docking Line*
Instead of merging changes directly into the codeline, merge them to a separate "docking" line where they are integrated and tested and then merged back to the codeline.

S5. *Staged Integration Lines*
Cascading branches are used to represent a hierarchy of promotion levels that correspond to points of integration and/or transfer of responsibility in the development lifecycle.

S6. *Change Propagation Queues*
Define change migration relationships between codelines (e.g. fixes from release 1.1 need to migrate to release 2.0) and ensure that changes originating from the same codeline are propagated in the same order in which they were originally completed and merged.

S6.1 *Auto-Propagation and Queuing*
When possible, propagations are performed automatically, otherwise they are queued-up for manual propagation.

S7. *Third Party Codeline*
When version controlling local changes to third party code, user separate codeline to represent the vendor's development and your own local development and merge updates from the vendor-line into the local-line.

S8. *Inside/Outside Lines*
Internal users of a centralized repository perform changes and integration on a restricted-access line reserved for internal-use only. Geographically dispersed users perform development on a separate external-line that is periodically synchronized with the internal-line.

Streamed Lines: Using the Patterns

We have presented a series of patterns for managing branching in parallel development projects. Certain subsets of these patterns represent conflicting styles and may not mesh well together for the same project; the patterns selected for a particular project are dependent on the needs of the organization and the project itself. In this section, we provide some guidelines on which patterns to select for your project. Which patterns you use will largely depend upon selected tradeoffs between safety and productivity (or "liveness"). More conservative strategies tend to tradeoff productivity for safety, while more optimistic strategies may do the opposite.

Generally speaking, using more branches for greater isolation reduces safety risks, but at the expense of more merging and integration effort. Using fewer reduces merging and integration efforts, but at the expense of less isolation and less safety. Merging sooner rather than later fleshes out risks early on while there is more time to address them, but requires continual efforts to regularly monitor and address such risks.

Select an Appropriate Branching Style

The first decision to make is whether to adopt the strategy of *Early Branching* or *Deferred Branching*. These are the two different "branching styles" underlying the majority of the branching patterns in *Streamed Lines*. *Early Branching* is better suited to larger or more formal efforts that require a high degree of fine-grained isolation and control; you assume less safety risks but pay the price of additional merging and propagation. *Deferred Branching* is good for projects that can afford to risk losing a bit of safety in order to gain more productivity; less branching and integration means less overhead, but also less isolation and verification.

The branching style that you decide is best suited for your environment will dictate a complementary set of patterns and pattern variants:

***Early Branching* Style**

- C2.2 *Branch per Change-Request*
- C3.0 *Codeline per Release*
- C2.0 *Branch per Task*

***Deferred Branching* Style**

- S1.2 *LAG Development Line*
- C3.1 *Codeline per Major Release*
- C2.1 *Branch per Major Task*

Regardless of the branching style selected, *Codeline Policy* and *Codeline Ownership* should used be for every branch and codeline created.

Patterns like *Parallel Maintenance/Development* and *Overlapping Releases* are typically the first branching structures many shops encounter. They can be applied using either branching-style. It depends primarily upon when you branch (early or late) and upon which effort goes on the branch and which stays on the parent codeline.

Early branching tends to keep the release or major release as the invariant for each codeline. So instead of splitting development and maintenance across codelines, it keeps the same release on the same codeline, regardless of whether or not it is development effort or maintenance effort for the given release.

For deferred branching, the releasing/maintenance effort will always be the one that branches off, allowing the latest and greatest development to continue on the same line as before. This way of thinking may be peculiar to those accustomed to an early branching style that uses separate codelines for each release; they may have difficulty understanding why it is coherent. With deferred branching, it's not the release that remains invariant on the branch, it's that the recency of the effort on the branch: the latest development efforts, or else the latest maintenance efforts.

Select Appropriate Merging Styles

Higher safety risk and less effort imply a relaxed policy toward codelines, and fewer integration lines; less risk implies stricter codeline policies, more codelines, and more integration effort. Although the choice of merging style often follows from the chosen branching style, a higher risk branching style does not necessarily imply a higher risk merging style. In fact, you may wish to offset low risk in one with high risk in the other. If you take more risks when splitting things apart, you may want to take less risk when putting things back together.

Remember that every time you add another line of integration, you are in effect, adding another level of indirection: you gain more isolation and nicer conceptual organization but you spend more time merging. It should be noted that

a *Virtual Codeline* is somewhat merge-evasive and may be used to simulate just about any kind of codeline. The merging patterns that are more suited to each branching style are as follows:

***Relaxed Merging* Style**

- P3.0 *Relaxed-Access Line*
- P6.0 *MYOC (Merge Your Own Code)*
- P6.1 *PYOC (Propagate Your Own Code)*
- S1.2 *LAG Development Line*
- C4.0 *Subproject Line*
- S4.0 *Docking Line*

***Restricted Merging* Style**

- P4.0 *Restricted-Access Line*
- P5.2 *Multi-Merge Early and Often*
- S6.0 *Change-Propagation Queues*
- S1.1 *Stable Receiving-Line*
- S5.0 *Staged Integration Line*
- S8.0 *Inside/Outside Lines*

In either case, frequent incremental integration is always a good idea (using *Merge Early and Often* or one of its variants) but the merging frequency and ownerships will differ between the two styles. The relaxed style favors liveness and assumes higher risk by having people merge and propagate their own changes across codelines. The more restricted style favors safety and has more codelines, each with more restricted access, and with codeline-owners performing most of the merges.

Unlike the branching styles, the merging styles may be mixed and matched to achieve a gradual progression from high-activity codelines with relaxed policies to lower-activity codelines with restricted policies. This can be accomplished with patterns such as *Docking Line*, *Subproject Line*, *Component Line* and *Remote Line*. But with a more relaxed style, each of these kinds of codelines will typically merge back to the development line while a more restricted style is more likely to use it as one in a set of *Staged Integration Lines*.

Start Simple

By choosing upon appropriate branching and merging styles, you have effectively decided upon risk management strategies for organizing and integrating work activities. Now you are ready to create some specific codelines. It is exceedingly rare for a single project to use *all* of the branching patterns presented here. The majority of projects will typically use the following "core set" of branching patterns (or one of their variants):

- *Mainline*
- *Codeline Policy*
- *Codeline Ownership*
- *Merge Early and Often*
- *Parallel Maintenance/Development Lines*
- *Overlapping Release Lines*

Take Baby Steps

Many projects will require little more than the above patterns, along with one of *MYOC*, *Docking Line*, or *Staged Integration Lines*. Other projects will have more sophisticated needs. They may start out with the above, and be okay for awhile; But they will eventually need to progress to the next tier of branching patterns, or their variants (often in the following order):

- *Codeline per (Major) Release*
- *Branch per (Major) Task*
- *Policy Branch*
- *Subproject Line*

Once again, one or more of the following merging patterns will be used with the above: *MYOC*, *Docking Line*, or *Staged Integration Lines*.

Evolving Integration Needs

Often, the project will take on more risk during early development and then gradually tolerate less and less risk as it grows in team-size, or moves more and more into maintenance mode. In addition to requiring more of the second-tier branching patterns above, merging styles may need to become less forgiving and more cautiously controlled:

- *Relaxed-Lines* may need to become *Restricted-Lines*

- The use of *MYOC* may need to change into *Docking Lines*
- *Component Lines* may need to be added for stable subsystems and modules
- *Docking Lines* and *Component Lines* may need to progress to a set of full-blown *Staged Integration Lines*.
- For projects where the number of merges or propagations can become unwieldy, *Multi-Merging* and *Multi-Propagating* may become necessary
- If you have a lot of change-propagations and want to stick with *MYOC* and *PYOC*, then you may need to use *Change Propagation Queues*

Special Project Needs

The following patterns are usually for "special needs" only:

- *Remote Line*
- *Third Party Line*
- *Inside/Outside Lines*
- *Platform Line*

You may need them very rarely or only for certain kinds of projects and project teams. But when the project does require them, they often have a very profound impact on the overall shape of the project-wide version tree, and on the overall organization of parallel development efforts. These patterns (along with *Change Propagation Queues*) should be used sparingly, and only as the need arises. This is especially true of platform-lines since it is often better to handle multi-platform issues with separate files and/or directories than with separate branches.

Revisit, Refactor, and Realign

As the project evolves, there will always be the need to periodically revisit, refactor, and realign the branching/merging structures adopted and their corresponding policies. You will also want look at the overall picture of the project-wide version tree and check to see if the tree looks too wide, too unwieldy, or too disjointed. Prudent use of codeline propagation and retirement into the *Mainline* will help guard against the tree becoming too wide. The patterns *Subproject Line*, and *Policy Branch* can help to correct a version tree that has become too complex and unwieldy. *MYOC* and *Docking Lines* can help remedy development that has become too isolated or disjoint.

Streamed Lines: General Advice and Recurring Motifs

The branching patterns in *Streamed Lines* don't cover every possible contingency. Situations will arise where the correct pattern or variant to use is not at all obvious, or may not even exist. However, even in these cases some of the recurring themes, which underlie many of the branching patterns presented here, may still be broadly applicable for your particular problem. These are as follows.

Use Meaningful Branch Names

Just like variable names in a program, each branch should have a meaningful name that communicates its purpose or its policy. If your VC tool doesn't directly support named branches, then *floating labels* (sometimes called *sticky labels*) can used to the same effect. See the pattern *Virtual Codeline*.

Prefer Branching over Freezing

Don't suspend all activities on a particular codeline when many of those activities could continue unobstructed on a separate branch, without impacting the efforts on the original codeline. See *Parallel Releasing/Development* for an example.

Integrate Early and Often

Frequent, incremental integration is one of the signposts of success, and its absence is often a characteristic of failure. Current project management tends to avoid strict waterfall models and embrace the spiral-like models of iterative/incremental development and evolutionary delivery. Incremental integration strategies like *Merge Early and Often* and its variants, are a form of risk management that tries to flush out risk earlier in the lifecycle when there is more time to respond to it. The regularity of the rhythm between integrations is seen by *[Booch]*, *[McCarthy]*, and *[McConnell]* as a leading indicator of project health (like a "pulse" or a "heartbeat").

Branch on Incompatibilities

Often, the best way to resolve opposing forces from competing concerns is create a new branch for the competition. Such incompatibilities may result from the following: access policies, dueling ownerships, integration frequency, activity-load, activity-type, and platform. Examples of this include *Policy Branch*, *Inside/Outside Lines*, *Component Line*, *Parallel Maintenance/Development*, and Platform Line.

Add Another Level of Integration

Sometimes branching on incompatibility isn't enough. Divergence will often require frequent convergence, or continuous mediation. In this case, it is often necessary to add another level of indirection, by adding another line of integration between the two opposing forces or competing codelines. Examples are *Subproject Line*, *Docking Line*, *Remote Development Line*, *Staged Integration Lines*, and *Mainline*.

KISS (*Keep It Simple Stupid*!)

Avoid branching hierarchies that are extremely wide or dense! (Think of "branch and bound.") Try for minimal reconciliation by creating new branches only when the added benefit is worth the added synchronization overhead. Use additional branches to provide greater isolation between tasks and changes; and use integration-lines to add additional verification and validation of merged changes.

But *don't use branches to solve all your problems*! Many problems are best addressed by different means. For example, numerous multi-platform issues are better solved by using extra files and directories rather than platform-branches. Don't use branches as a "hammer" to make every problem look like a nail, and don't "sow" a new branch unless you can reap the benefits.

Preserve Integrity and Consistency

Preserve the conceptual integrity of the branch! When delegating volatile aspects of high-impact variation to separate branches, keep each aspect logically consistent within its own branch: keep codeline usage consistent with its policy, and keep codeline policy consistent with its purpose. Occasional "fine-tuning" and remedial actions are to be expected, but avoid changes that violate the spirit of the codeline's intent.

Preserve the physical integrity of the branch! Don't merge incomplete or inconsistent changes into the codeline; and don't leave codelines in inconsistent states. When the configuration of a codeline is inconsistent or incorrect it can adversely impact all users of the codeline. Try to keep codelines reliably consistent, and consistently reliable.

Choose optimistic or pessimistic branching policies and stick with them! For a given project, strike a sensible balance of trade-offs between safety (isolation, access control, code integrity, and risk mitigation) and liveness (productivity, integration overhead, working "on-line") and then apply them in a consistent manner. The balance may need to be dynamically adjusted over time; but at any given time, the policies should be consistent with one another.

Isolate Change

You may recall that one of the recurring themes in the *[GoF]* **Design Patterns** book is: "Encapsulate the thing that varies." Branching doesn't achieve encapsulation of information so much as it achieves isolation of changes. So a recurring theme in most of these branching patterns is "*Isolate the thing that varies*!" Each branch and codeline isolates one or more of the following dimensions over a given time-period:

- Physical Structure - organization and distribution of:
  - System knowledge
  - Components and subsystems
  - Configuration elements (files and directories)
- Functional Evolution - organization and distribution of:
  - Change and change-flow
  - Delivery (releases and patches)
  - Functionality (requirements, features, fixes, and enhancements)
- Teamwork - organization and distribution of:
  - Interaction and communication (coordination and collaboration)
  - Policy and procedure (cooperation and control)

  - Workflow and activity-flow
  - Roles and responsibilities
- Environment and infrastructure (platform and resource variations)
- Reproducibility and traceability (identification and tracking)

Isolate *Work*, not People

Perhaps most importantly, the branching policies and patterns described here do *not* remove the need for communication between project team members. These patterns should facilitate communication, not eliminate it! The goal of these patterns is to help isolate *work*, not people. People working together on a project need to remain socially connected and coordinated with one another, and maintain an awareness of the impact of their efforts downstream and throughout the entire lifecycle. Jeopardize this and you jeopardize team synergy, and ultimately, team success.

If you isolate people from their work, systemic disconnection may result: developers lose touch with the effects of their own efforts on the overall project. If you segregate people from each other according to their work tasks, social isolation may occur: people lose touch with one another and with the overall project team. The purpose of parallelization is not to isolate people from people, or people from their work, but to isolate work from *other work*. Conway's Law (see [Cope95]) applies just as much to the architecture of the project's version tree as it does to the architecture of the system. Use this wisdom to your advantage (and ignore it at your peril).

Streamed Lines: Resulting Context

Similarities with Concurrent/Parallel Programming

[McKenney95] writes of the forces for and against parallelizing a software program, breaking them down into: Speedup, Contention, Overhead, Economics, Complexity, and a few others. Most of these forces are equally applicable to the case of concurrent/parallel software development. In fact, designing parallel development strategies for concurrent software development bears more than a striking resemblance to parallel programming strategies for concurrent object systems. The former deals with multiple collaborating objects running in multiple threads of execution across multiple address spaces in a parallel software program; the latter deals with multiple collaborating individuals working in multiple threads of development across multiple workspaces in a parallel software development project.

As [Lea96] describes, some of the most basic tradeoffs to be made when designing concurrent object systems are those of *safety* ("The property that nothing bad ever happens") and *liveness* ("The property that anything ever happens at all"). These tradeoffs are essentially the same for software development:

> From either direction, the goal is to assure liveness across the broadest possible set of contexts without sacrificing safety.

The need to apply such strategies across the broadest possible set of contexts ties into their reusability across the project, and between projects. Hence all the same issues and concerns mentioned by [Lea96] regarding safety, liveness, and reusability also arise during parallel development.

Isolation and Risk Mitigation

Branching is an optimistic concurrency control strategy for parallel development. It tries to mitigate the risk associated with such optimism by separating concurrent/parallel efforts into isolated paths of development. Branching off into separate workstreams is fairly easy to do with minimal interference, and gets rid of the need for development tasks to "block" waiting for checkout-locks to be released. Rejoining the two paths after they've been separated is done via integration (merging). The inherent risk in resynchronization is mitigated by allowing it to happen in a well-insulated context at a more convenient time.

In effect, *every codeline and branch represents a form of risk management* by isolating how functionality, environment, knowledge, teamwork, responsibility, and reliability, are distributed and disseminated across time and space.

Managing Complexity with Hierarchy

Branching and merging hierarchically decompose and recompose parallel development into more manageable chunks! By isolating things along various dimensions in a hierarchical fashion, we are attempting to manage dynamically evolving complexity and dependencies. First we decompose the parallel development problem into codelines and branches and subbranches, then we recompose the subparts back into the larger whole by progressively merging subbranches back to branches, branches back to codelines, and codelines back into the mainstream.

Integration Overhead

Regardless of whether changes are reconciled and synchronized immediately, or deferred to a more convenient time and place, there is always a risk of compromising the integrity of the codeline after a merge. That is the price for this kind of optimism. The usual laws of thermodynamics (regarding entropy and enthalpy) apply here as well: it is usually harder to put things back together than it was to take them apart. *For every branch created, there is almost always an opposing merge to be reckoned with!*

Integrity and Reproducibility

By separating development into isolated development paths and change-tasks, branching eases the burden of tracing changes (both physical and functional) and their dependencies. This makes configurations, features and faults easier to verify and reproduce. Although each merge carries with it some additional risk to codeline safety, intelligent use of branching and merging really can help to preserve the conceptual integrity of the codeline, as well as its physical integrity (ensuring configurations are correct and consistent).

Communication, Coordination, and Productivity

If your VC tool supports symbolic branch names (rather than numeric ones) then mnemonic branch names can serve as an effective form of communication that describes the intent of the branch and the work that takes place upon it. If you aren't using such a VC tool you may need to find a way to work around this, either using a technical solution (like *Virtual Codeline*) or a social convention among the project team.

Branching also helps communication and collaboration be effectively organized, synchronized, and parallelized. If used properly so that it isolates work instead of people, branching promotes effective teamwork and really can reduce time-to-release. If you make the effort to apply intelligent, risk-aware strategies for the selection of branching and merging styles, and periodically take a step back to review and revise the project-wide version tree, you should be able to reap the benefits of parallel development (shorter cycle-time) and keep the amount of synchronization overhead (and risk) to a manageable level.

Parallelization with Concurrency Constructs

Despite the fact that many VC tools consider branching to be one of their nicer and more advanced features, branching is in fact a somewhat low-level construct used for concurrency control. It is not ideally suited for parallelization of work and workflow at coarser-grained levels beyond a single file or directory. Using branching for this purpose results in a non-trivial amount of trivial merging where revision contents need to be propagated from branch to branch with little or no difference between them. Good merging tools can minimize the pain and overhead associated with this, but if the overhead can still be noticeable.

The problem is that the majority of readily available VC tools don't provide the user with anything better. It would be far more suitable if one's VC or SCM tool provided predefined constructs which directly map to the conceptual notions of: change-sets, activities, and activity-streams, without being dependent upon branches. Then we could use the SCM tool to directly model parallel effort and workflow and let the tool itself worry about how to handle the low-level concurrency control (branching) with the help of some user-supplied policy preferences. There are a select few tools which actually do provide this capability (ClearGuide is one of them) but they are presently in the minority. So unless you are using such a tool, branching tends to be the next best mechanism for supporting parallelism.

Branching Topology Comprises Workflow

The result of using all these branching patterns is a version branch tree structure that, for the most part, represents the intended structure of activity workflow for the project. One might regard this as a simple byproduct of *Conway's Law*, namely that "Architecture follows Organization" (see [Cope95]). In the case of branching for parallel software

development, we might rename this as a corollary to Conway's Law and call it "*Branching Topology Comprises Workflow*."

Architectural Alignment

Software development involves a great many architectural structures (or "views"). Most of us are familiar with the logical design view of software that most commercial modeling tools support. But there are other structural views for other aspects of the project or product that encompass issues such as: platforms, domains, development processes, interfaces, resources, configuration, reflection, extension, generation, analysis, usage, and even social dynamics. Both [Kruchten95] and [Davis97] describe "4+1 Model Views" of software architecture with five architectural views; [Kriha97] proposes no less than ten different architectural structures for software which can be difficult to visualize, yet dangerous to ignore.

The larger a project and its development team, the more complex each view or structure will be (to say nothing of the dependencies and relationships between structures). Part of successful project management and product architecture lies in keeping these different structures in reasonably close alignment with one another. This minimizes the complexity of the overall architecture and makes the structures themselves easier to manage. Greater structural similarity between views means greater conceptual integrity and coherence between views and in the overall project and product.

The branching tree of a project represents the structure of its evolution in terms of change-flows. The flow of work activities is also an important project structure. *Streamed Lines* attempts to coordinate these two sets of structures so that activity and workflow conveniently map to change-flows (using branches as the grouping mechanism). This helps makes the project's development and evolution easier to conceptualize and manage. In this manner, *Streamed Lines* assists in bringing some of the architectural structures of a software project into alignment.

Appendix A - *Streamed Lines*: The Patterns

- Branching Policy Patterns
- Branch Creation Patterns
- Branch Structuring Patterns

Branching Policy Patterns

- P1. *Codeline Policy*
- P2. *Codeline Ownership*
- P3. *Relaxed-Access Line*
- P4. *Restricted-Access Line*
- P5. *Merge Early and Often*
- P6. *MYOC (Merge Your Own Code)*
- P7. *Early Branching*
- P8. *Deferred Branching*

P1. Codeline Policy

Pattern ***Codeline Policy***

Aliases *Policy per Codeline*

Context You are developing software within a system that utilizes multiple codelines.

Problem How do the developers know which codeline to save their code in, and when to save it?

Forces

- Each codeline has a different purpose; one codeline might be intended for fixing bugs in a particular release; another codeline might be used for porting existing code.
- A codeline's name is usually indicative of its purpose;
- A codeline's name usually doesn't express all the finer points of codeline usage;
- If code is written to the wrong codeline, the bad changes must be backed out, causing productivity to suffer.

Solution In addition to using naming conventions and meaningful branch/codeline names, formulate a coherent purpose for each codeline. Describe the purpose in a clear and concise policy. The policy should be brief, and should spell out the "rules of the road" for the codeline, including:

- The kind of work encapsulated by the codeline, such as development, maintenance, a specific release, function, or subsystem;
- How and when elements should be checked-in, checked-out, branched and merged;
- Access restrictions for various individuals, roles, and groups;
- Import/export relationships: the names of those codelines it expects to receive changes from, and those codelines it needs to propagate changes to;
- The duration of work or conditions for retirement of the codeline;
- The expected activity-load and frequency of integration

Keep in mind that not all codeline policies will require all of the above information. Only specify what is needed. Some VC tools allow you to associate a comment with each branch and codeline name. This is an ideal place to store the description of a suitably brief codeline policy. Developers can run a command using the codeline name instead of digging elsewhere for documentation. Otherwise, store the codeline policy in a well-

known, readily accessible place (and perhaps provide a simple command or macro that will quickly display the policy for a given codeline name).

Resulting Context The codeline now has a coherently communicated purpose that is readily available to the entire project team. This sets the stage for how the codeline can and should be used. Codeline access issues are resolved by determining whether or not they violate the intended purpose of the codeline.

Related Patterns *Codeline Ownership* and *Policy Branch* describe tactics for resolving policy conflicts. The owner of the codeline policy document should be the current owner of the codeline. *Restricted-Access Line* may be applicable if the codeline policy calls for tight permission or control. Every branch creation and branch-structuring pattern will require a corresponding codeline policy for each codeline.

P2. Codeline Ownership

Pattern ***Codeline Ownership***

Aliases *Branch Ownership*

Context You are a developer working with at least one codeline in a set of Multiple codelines. A *Codeline Policy* has been used to define policies for codeline checkin/checkout. Someone wants to do something to a codeline that its policy doesn't seem to address, or else the policy appears vague regarding this particular issue.

Problem Should the action affecting the codeline be performed or not? How can this be decided while at the same time ensuring the integrity and consistency of the codeline?

Forces

- "In theory, practice and theory are the same, but in practice, they're very different." No single policy can spell out every situation that can arise: *Codeline Policy* covers theory, but something else is needed to cover practice.
- If the *Codeline Policy* is unclear, the developer will need to have the policy clarified.
- A codeline policy can be violated, either accidentally or intentionally.
- The codeline needs to stay in a correct and consistent state to avoid adversely affecting "in progress" development tasks.

Solution Assign one owner per codeline. It is the owner's responsibility to

- Clarify the codeline if it is found to be unclear;
- Decide whether to leave in or back-out files that have been checked into the codeline in violation of policy;
- Make rulings on codeline postings in ambiguous or inapplicable situations;
- Assist in, or perform, the integration of changes to the codeline;
- Decide when the codeline should be frozen and unfrozen, and when it should be completed and merged back to the *Mainline*.

Ownership does not necessarily imply exclusive access, but it does imply authoritative access control. Perhaps only the codeline-owner may check in files (*Restricted-Access Line*); or perhaps others can checkin files if they obtain approval for the codeline owner before checkin, or notify the codeline-owner immediately after checkin. (*Relaxed-Access Line*). The Codeline Policy should clearly define the ownership style for access-control

style. In general, frequently accessed codelines with many developers will require a more restrictive ownership policy than an infrequently accessed codeline. So will codelines that encompass high-risk or high-complexity activities, or that demand high levels of reliability. Smaller projects and teams for codelines encompassing less critical tasks can afford more permissive codeline ownership and access control policies while still maintaining the integrity and consistency of the codeline.

Variants

P2.1 ***Codeline Dictatorship***

An extremely restrictive form of codeline ownership whereby checkouts and subbranches of a branch are restricted, in addition to checkins. The dictator may be a single person, or an entire site. A common example is *a Remote Development Line*: remote developers might be prevented from creating new versions and branches that do not originate from the remote branch. Thus, the local branch is reserved exclusively for the "master" development site. This is essentially how ClearCase Multisite defines the notion of "*branch mastership*."

Resulting Context

- A single individual is held accountable for the consistency and integrity of the codeline. As a result, the codeline is more likely to be in a consistent and reliable state.
- Holding the owner of the codeline responsible for its state, decreases the likelihood of the codeline policy being trampled on, or of the codeline being used for the wrong purpose.
- The conceptual integrity of the codeline is maintained by having a single mind, the codeline owner, serve as a single point of authority for resolving codeline issues.

Related Patterns

A *Codeline Policy* should specify the usage rules that codeline-owner is expected to ensure. Branch locking may be used to impose *a Restricted-Access Line* enforcing mastership or ownership policies. *Code Ownership* ([Cope95]) will apply for codelines serving as component branches or component lines; the code and codeline owners should be the same. Similarly, *Owner per Deliverable* ([Cockburn97]) should be applied for functional branches and subproject-lines;

P3. Relaxed-Access Line

Pattern ***Relaxed-Access Line***

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

- If there are many developers working on codeline, or some of them are inexperienced, a tighter rein may be necessary.
- If there is a significant level of risk or complexity associated with the type of work taking place on the codeline, checkins and merges may require close monitoring and/or verification.
- It is important to ensure the codeline is always in a consistent state so as not to adversely impact the people working on it.
- If the codeline corresponds to a certain promotion level (see *Staged Integration Lines*) or lifecycle phase, then this may dictate what lengths are necessary to verify the consistency of this particular codeline.

Solution If the codeline is being used for development or maintenance (rather than exclusively for integration) and if the group working on the codeline is relatively small, experienced, and reliable, then give the developers relatively free reign to do what they already know how to do: work together to develop software in a timely fashion. Use the bare minimum of checks and controls but make sure the codeline-owner takes his job seriously and is always aware of the state of the codeline and whether or not its integrity is about to be compromised by a particularly risky/complex development task.

Variants

P3.1 ***Guarded Merge Access***
One way of addressing possible merge/integration conflicts is to let developers checkin to the codeline as they see fit, but to perform an automated check ensuring there will be no merge conflicts upon checkin. Some VC tools provide triggers as a convenient way of performing such checks. A pre-checkin or pre-submit trigger can invoke the VC tool's automated merge tool in "what-if" mode and abort the operation unless the merge is trivial or has no conflicts at all. If your VC tool doesn't have triggers then you would typically use a "wrapper" instead, wrapping your own command or script around the checkin/submit command (or else you could always have the codeline-owner perform the merge-query). If your VC tool doesn't have such nice merge facilities, you would need to put that together on your own, using history information stored in the repository to build and traverse the version tree for a given file.

If the merge tools think the merge is trivial, then the checkin/submit proceeds. Otherwise, the person who made the changes is required to re-sync their changes with the latest state of the codeline (importing any recent codeline changes into their workspace) and resolve the merge conflicts in their own workspace before attempting to check them in to the codeline again.

Resulting Context The codeline has a more *laissez-faire* access policy that trades off safety for liveness. This is more likely to be an acceptable risk for a small, experienced group of developers. While "glitches" may have a higher impact on the codeline, their likelihood is decreased by the more trustworthy nature of the group and the work they are performing. When glitches do occur, the group working on the codeline is often intelligent enough to quickly contain and correct the problem.

Related Patterns *MYOC (Merge Your Own Code), Merge Early and Often*

P4. Restricted-Access Line

Pattern ***Restricted-Access Line***

Aliases *Branch Locking*

Context Same as for *Relaxed-Access Line*

Problem Same as for *Relaxed-Access Line*

Forces Same as for *Relaxed-Access Line*

Solution If the codeline will have a large number of developers (or a sufficient number of inexperienced developers) and the work taking place on the codeline is very risky, or consists primarily of integration, or corresponds to a promotion-level or lifecycle phase that is not very close to the initial level, then restrict who is allowed to checkin code on the

codeline to either just the codeline owner, or a to a small set of trusted, reliable developers.

Some VC tools support access control for branches so that they may be locked against checkin or checkout. Sometimes the access control will allow you to specify everyone, and/or a list of users or groups to include/exclude from the codeline access-list. If the VC tool you are using has these features, they prove quite convenient for enforcing codeline access-restrictions. If the VC tools don't support such features, the triggers or wrappers for checkin/checkout will be required (and they will need to determine, or else query the user for, the target codeline).

Variants

P4.1 ***Codeline Freeze/Retirement***

It is common to suspend ("freeze") development to perform release integration and engineering activities. Afterward the codeline or branch may be decommissioned, sent into "retirement" by cutting it off and merging it back to a parent codeline or to the *Mainline*. Or development on the codeline may be resumed ("thawed" or "melted") after the release has been "cut." During the time it is frozen however, no one may make any changes to the codeline that aren't deemed necessary for successful integration and release. The only codeline activities permitted are those which sync and stabilize its contents.

When using branches, you may continue development on one codeline while conducting release engineering on another (see *Anti-Freeze Line*). You may still want to freeze any new changes (other than integration) on the codeline that is being prepared for release, but remaining development activities can carry on in separate codelines.

P4.2 ***Export Lock (a.k.a. Integration Lock)***

When developers use *MYOC* to export their changes to the codeline, there may be contention between multiple developers trying to commit their changes for export during the same time period. Just before a completed change is about to be merged back into the codeline, the developer acquires an export lock. No one but the holder of the export lock may checkin to the codeline during that time. Also, the developer may only checkin all their changes at once, or none at all (logical changes to the codeline must be "atomic").

If the merge can be performed quickly and cleanly, and its correctness can be verified shortly thereafter, then the developer keeps the export lock and does not have to reconcile changes against a moving "target." If retaining the export lock during the merge would adversely impact others, the developer relinquishes the export lock and reconciles with the current state of the codeline (allowing others the opportunity to export their changes). When the developer is once again ready to export, there may now be new codeline changes to contend with, and the same decision of whether or not to keep or relinquish the temporary export lock will be faced once again.

P4.3 ***Private Line (also Private Branch)***

Sometimes a codeline or branch is primarily intended for a single person to perform a single task or series of tasks. In these cases it is common the prevent all users other than the codeline owner from checking-out (and in) files from the codeline. Sometimes even branching off the private line is forbidden to all but the codeline owner.

Also, if a codeline is used as a *Component Line*, and if only the code-owner is allowed to checkin the code she owns, it makes good sense to restrict all access on the component-line to the code-owner (who also serves as the codeline owner, of course).

P4.4 ***Role-based Line Access (Role Branch)***

Sometimes, instead of being restricted to one or more users, a branch is restricted to one or more development *roles*. Any person may use the branch or codeline as long as they are playing the correct role at that time. This is typically done using a special user-id or group-

id for each development role requiring special access. For example, in order to gain merge access to a codeline, it might be the case that a developer fulfilling an integration role must first become user '`integratr`' or be a member of the '`integratr`' user-group.

Resulting Context Restricted-access provides an added level of safety and security for the codeline. If the codeline is shared, then you lose some of the benefits of choosing file-branching over file-locking (resulting in more waiting time), but the restriction is larger-grained and not absolute; so you still keep many of the benefits of branching versus locking while ensuring that efforts on the codeline remain as isolated and/or as consistent and stable as demanded by the nature of the codeline.

Related Patterns *Codeline Policy, Codeline Ownership, Remote Development Line, Inside/Outside Lines*

P5. Merge Early and Often

Pattern ***Merge Early and Often***

Aliases *Codeline Update per Task*

Context For a given codeline, maintenance and/or development activity is regularly occurring. Some of the work may be happening directly on the codeline, while other tasks ultimately intended for this codeline may be taking place on *Task Branches* or *Subproject Lines*. Furthermore, the codeline may also require periodic synchronization with changes on *Remote Lines* and changes often need to be propagated from a maintenance-line to a release-line, or from a release-line to a codeline for a subsequent release.

Problem When should changes be merged into the codeline and how frequently should they be merged?

Forces

- You want to preserve a consistent, correct state of the codeline
- Frequent merging may upset the stability of the codeline, making it volatile, and hard for developers to stay in sync
- If developers are employing dynamic configuration selection, then changes to the codeline are immediately reflected in their workspace, and may upset their own work
- Sound risk management dictates identifying and addressing issues as early as possible, and merging can certainly bring significant issues to light that might not otherwise be noticed until a deadline looms near
- The more out-of-sync developers are with respect to the latest state of the codeline, the more likely it is that their changes will have non-trivial conflicts that are time consuming to reconcile when it comes time for their changes to be incorporated into the codeline.

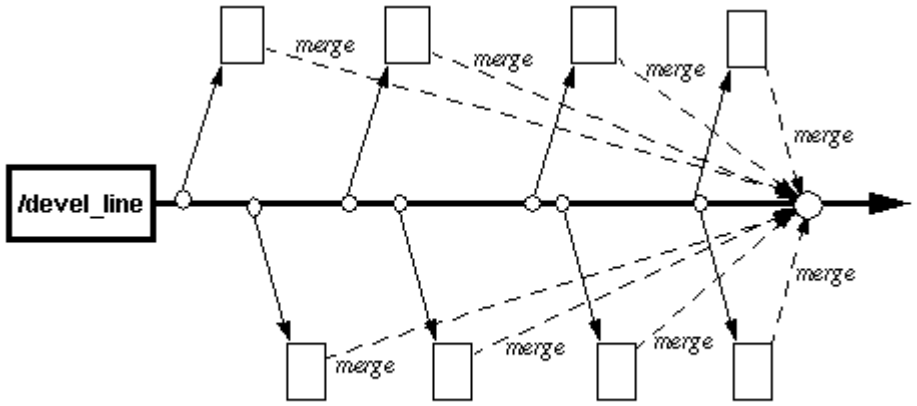


Figure P5a: Big-bang integration at the end (undesirable)

Solution Integrate new changes into the codeline as soon as they are ready; Do not integrate portions of a logical change before the entire change completed. But once the entire change has been completed (much like a transaction) it should be immediately considered for integration into the codeline and merged in as soon as conveniently possible.

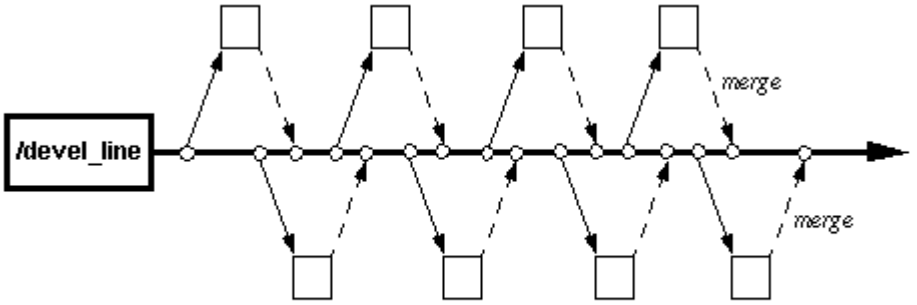


Figure P5b: Merging back to the codeline after each activity-branch

Variants

P5.1 ***Propagate Early and Often (a.k.a Propagation per Task)***
This is basically the same thing but it refers specifically to the case where changes already integrated (or merged) into one codeline need to be propagated (merged) into another codeline.

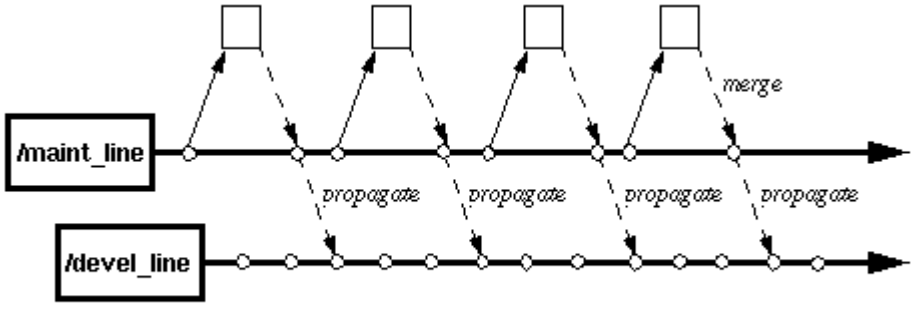


Figure P5.1a: Frequent Propagation for Parallel Maintenance/Development Lines

P5.2 ***Multi-Merge Early and Often***

If the codeline owner must be responsible for all merging (due to the nature of the codeline requiring a high degree of reliability and consistency that only the line owner can take responsibility for), then, for a high-volume (high activity-load) codeline, it may not always be feasible to merge every change-task as soon as it is complete.

In this case, merge-tasks need to be queued or batched up into manageable chunks and then regularly merged in. A suitable interval should be chosen based upon the number of tasks and their average complexity. Daily might be best for high activity codelines, whereas as weekly will be better for others. Start off trying for daily (or nightly) multi-merges (tested via Daily or Nightly Builds). If that is too burdensome then try semi-weekly and then weekly. If a shorter time period is needed than try twice a day (daily & nightly). It will take a little while to find the correct rhythm, and it may fluctuate form time to time, but it does need to be done as frequently and regularly as manageable and should be done at least weekly.

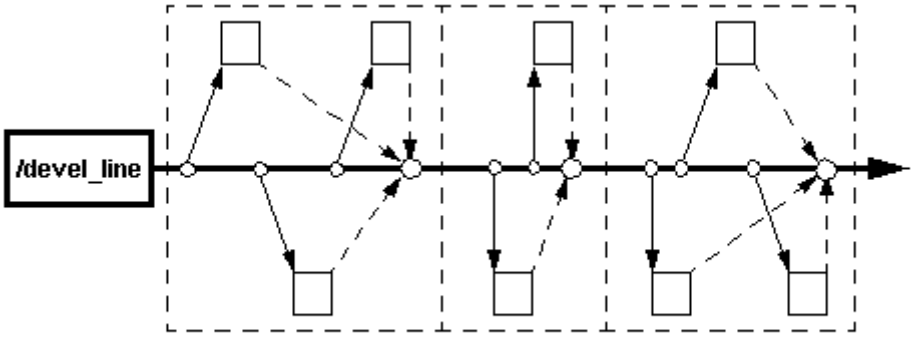


Figure P5.2a: Multi-Merging back to the codeline after a few activity-branches

This may be combined with change-propagation to yield "*Multi-Propagating*":

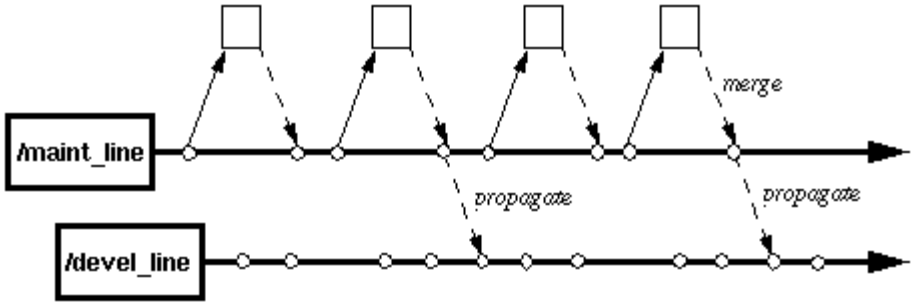


Figure P5.2b: Multi-Propagating for Parallel Maintenance/Development Lines

Resulting Context

The codeline is incrementally updated at fairly regular intervals to reflect the current state of progress for that codeline. Subsequent development on that codeline now takes place in the context of the newly merged changes. While this may introduce some difficulty if a change later needs to be backed out, it serves as a forcing function to help identify and flesh out risks early in the development cycle when their is more time and opportunity to address them intelligently.

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.

Related Patterns *MYOC (Merge Your Own Code), Change Propagation Queues* follow naturally from this pattern. *Branch per Task* may be suitable for developers using dynamic configuration selection. *Floating Labels* (see *Virtual Codeline*) may also be a good alternative.

P6. MYOC (Merge Your Own Code)

Pattern ***MYOC (Merge Your Own Code)***

Aliases *Get the right person to do the merge!*

Context You are a developer working on code in either a branch off of the codeline or on the codeline itself. After you have finished debugging and testing your changes, you wish to follow *Merge Early and Often* and have your changes merged back into the codeline.

Problem Who should perform the merge, and who assumes the burden of ensuring it is integrated correctly?

Forces

- At the time of the merge, the contents of the codeline may be different from its contents at the time the change began.
- The person performing the change-task (the "change-owner") may be different from the code-owner for the changed files (see *Code Ownership* in [Cope95]). Both of these people may be different from the codeline-owner.
- The change-owner or the code-owner usually knows the most about the code that was changed. High-complexity and high-risk changes increase the importance of being familiar with the changed code.
- If the codeline changed significantly while the developer was performing the change-task, the change/code owner may be unfamiliar with the current state of the codeline.
- The codeline-owner usually knows the most about the codeline's present state. Codelines that must be depended upon for high stability/reliability increase the importance of being familiar with the current state of the codeline.
- Systemic disconnection: Developers who aren't responsible for merging their own code back into the codeline can easily lose sight of the impact of their efforts further "downstream."
- Social isolation: Developers who simply take their change and "throw it over the wall" can lose sight of the needs of other team members on the other side of the "integration wall."
- You don't want the codeline falling into an inconsistent state, which adversely affects its users, as a result of the merge.

Solution If either the change-owner or the code-owner also happens to be the codeline owner, then have the codeline-owner merge the code. Otherwise, unless reliability and process control obligations are so strict as to forbid merging by anyone but the codeline-owner, always strive to have the either the change-owner or the code-owner perform the merge.

On occasion, this will require cooperation and collaboration from all three parties, especially the codeline-owner if the codeline-state is higher risk or has significantly changed. Remember that a change to merge may conflict with a previous change in the

codeline. Resolving this conflict may require additional assistance from the person who performed or merged the conflicting changes.

Hopefully, it will be the case that the change-owner and the code-owner are one and the same, or else that the code-owner and the codeline-owner are the same person (in which case the codeline may be a *Component Line*). If all three of these owners are different people, you may need another level of integration (see *Staged Integration Lines*).

Otherwise, if the codeline is necessarily restrictive or the degree of risk/complexity for merging to the codeline is very high, the codeline owner may have to perform the merge. In this case a better solution may be (again) adding another level of integration to create a *Docking Line*.

Variants

P6.1 ***PYOC (Propagate Your Own Code)***
If change-tasks are propagated on a task-by-task basis, then pretty much the same scenario occurs when it is time for a change-task that has been integrated in one codeline to be propagated to another codeline. So now we have two codeline owners, the owner of the receiving codeline and the owner of the sending (or originating) codeline. Despite this difference, the rule of thumb is still pretty much unchanged.

Resulting Context Developer's take responsibility for their code and changes throughout the codeline's life span instead of playing "pass the buck" and becoming alienated from each other's efforts and responsibilities. The codeline owner still needs to monitor/verify the codeline after merges, but isn't overburdened with a high-volume of merges to perform.

Related Patterns *Codeline Ownership, Code Ownership* [Cope95], *Merge Early and Often*

This pattern requires a sufficiently *Relaxed-Access Line*. If that is not feasible then a *Docking Line*, or *Staged Integration Lines* may be a good compromise that allows developers to merge their own changes to another codeline which eventually cycles those changes (under more protective jurisdiction) back into the development line.

P7. Early Branching

Pattern ***Early Branching***

Aliases *Branch Early and Often*

Context During project planning and later in course of development efforts, you become aware of various tasks and subproject that will best be performed in parallel with other development efforts. These might be major features or fixes, major or minor releases, patches, or new platforms (or even geographically distributed development).

Problem When should you create new branches or codelines for parallel tasks and subprojects that are forthcoming in the near future?

Forces

- For every new branch that is created, there is the risk that it will require a *greater than* equal and opposite merge
- You want to avoid the project version-tree being too overwrought with concurrent branches
- You would like to isolate any parallel tasks that may potentially conflict with one another

- You want each new codeline and branch created to have a coherent purpose that fulfills a legitimate need

Solution Create new branches and codeline as soon as the corresponding parallel work efforts begin. Even if they don't yet conflict with any current work on existing codelines, separate those activities into logically separate branches which are later merged back into parent codelines.

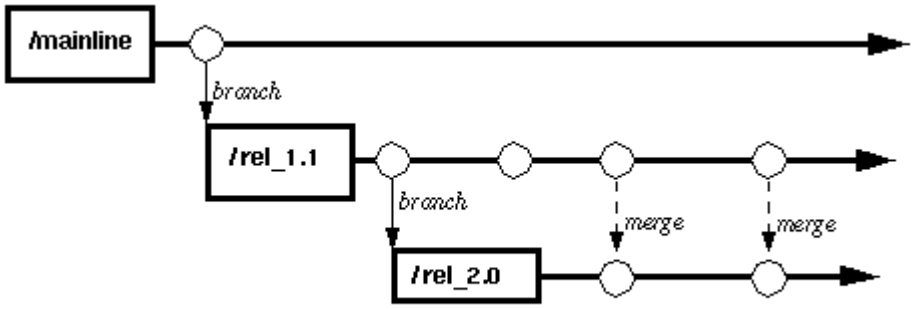


Figure P7a: Early Branching for releases 1.0, 1.1, and 2.0

Resulting Context Each branch and codeline has a logically coherent purpose that is conceptually clear and easily maps to tasks, subprojects, features and releases. Although these activities might have easily occurred on the same codeline without any conflict, each branch helps to isolate logically distinct efforts and to encapsulate a cohesive unit of functionality, effort, or delivery.

Related Patterns *Branch per Task, Codeline per Release*

P8. Deferred Branching

Pattern ***Deferred Branching***

Aliases *Lazy Branching, Late Branching*

Context Same as for *Early Branching*

Problem Same as for *Early Branching*

Forces Same as for *Early Branching*

Solution Instead of spawning off a new task branch, subproject line, or release line, as soon as effort begins for the corresponding goal, wait to start the new branch until work for that release actually starts to conflict with work that is already on the codeline. This doesn't mean that you should wait to branch until one or more files have conflicting changes. What it means is you should wait to branch until a logical change (feature, function, or bug-fix) that is needed only for the new task, and not yet desired in the original codeline, is about to take place.

For releases, perform development for the new latest-and-greatest release on the *LAG Development Line* until it is no longer the latest release effort (then branch off into a major release-line as in *Codeline per Major Release*). When work is first ready to begin for the new release-line, instead of starting a new branch for the new effort, you start a new branch

for the *old* effort (that was, up to now, taking place on the LAG-line).

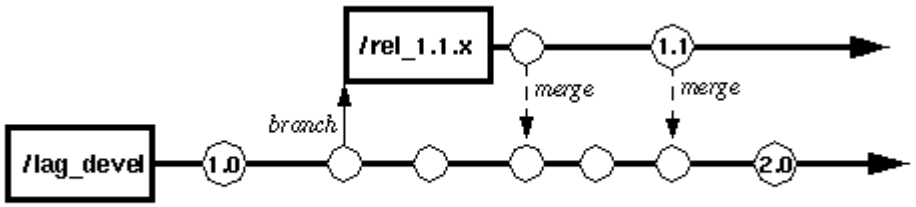


Figure P8a: Deferred Branching for releases 1.0, 1.1, and 2.0

Similarly, with features and fixes, if they would need to be merged back to the original codeline by release-time anyway, and aren't of a significantly high risk/complexity or long duration, then go ahead and perform them on the codeline (see *Branch per Major Task*).

Resulting Context

Deferred Branching requires less integration/merging and propagation of changes from codeline to codeline. It does so at the risk of less isolation, trading off "safety" for "liveness".

The branches that are created may seem less "singular in purpose" than with *Early Branching*. With *Early Branching* the type of work for each branch/codeline happened only on that branch. With *Deferred Branching* the work happens first on the original codeline, and than later gets branched-off to another codeline. This may seem inconsistent or cause people conceptual difficulty in understanding and remembering what kind of work should be occurring on the codeline. Sometimes this can be addressed by adopting a slightly different perspective for the purpose of the codeline.

For example, instead of defining the purpose of the codeline to be for a specific release, define it to be for the latest and greatest development effort of the project (*LAG-Line*) of for the latest and greatest maintenance efforts for a previous release (*Codeline per Major Release*). However, this conceptual reorientation this cannot always be accomplished in a suitably coherent and consistent manner, and even when it can, some team members may have significant difficult making the adjustment.

Related Patterns

LAG Development Line, Branch per Major Task, Codeline per Major Release

Branch Creation Patterns

- C1. *Policy Branch*
- C2. *Branch per Task*
- C3. *Codeline per Release*
- C4. *Subproject Line*
- C5. *Virtual Codeline*
- C6. *Remote Line*
- C7. *Component Line*
- C8. *Platform Line*

C1. Policy Branch

Pattern *Policy Branch*

Aliases *Branch on Incompatible Policy*

Context A number of developers are working with the same codeline. A *CodelinePolicy* is being used with the codeline.

Problem Some of the users are dissatisfied with the current codeline policy, and have a need for it to change.

Forces

- Changing the *CodelinePolicy* to the policy requested by the dissatisfied users will cause problems for the other users of the codeline.
- Adding new policy statements to the existing policy may make the codeline unusable in any form.

Solution Branch the codeline to create a new one based off of it; leave the old *CodelinePolicy* with the original codeline and attach a new *CodelinePolicy* to the new codeline.

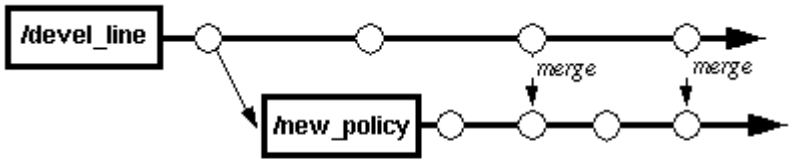


Figure C1a: A Policy Branch spawned from a Development Line

Resulting Context

- Developers on the original codeline, and on the new policy branch each have a policy more suited to their needs.
- Additional integration needs to take place to keep one of the two codelines "in sync" with the other via change-propagation.

Related Patterns Don't forget to use *Codeline Ownership* and *Codeline Policy* for the new policy branch. *Propagate Early and Often* to keep the codelines "in sync" as needed.

C2. Branch per Task

Pattern ***Branch per Task***

Aliases *Activity Branching, Task Branching, Side Branching, Transient Branching*

Context For a given codeline, the set of files affected by any change-task may overlap with that of other changes to the codeline. At some point, all the features and fixes the codeline will need to be integrated together in order to deliver the required combination of functionality.

Problem How can multiple, potentially interfering or overlapping changes be made to a codeline without compromising its consistency and integrity?

Forces

- Parallel development without controlled interaction between concurrent changes can result in lost or corrupted changes, wasted effort, and rework.
- Concurrent changes need to be integrated with care. Otherwise changes which work correctly in isolation from one another could cause incorrect or inconsistent behavior when combined together. A single unanticipated change could destabilize the entire codeline and impact all of its users.
- Excessive locking may cause very long "busy waits", forcing developers to suspend some of their activities until a locked file becomes available. At its worst, this can cause deadlock between parallel activities.

Solution Fork off a separate branch for each activity that effects changes for a codeline. Once the activity is complete and all of its file changes have been tested, checkin files on the activity branch. Then, if *MYOC* is being used, merge the activity branch into the appropriate codeline (as a single transaction). Otherwise notify the appropriate codeline owner that the change is complete and ready to be merged (and provide the codeline-owner with any other necessary information for finding, merging, and testing the change-task).

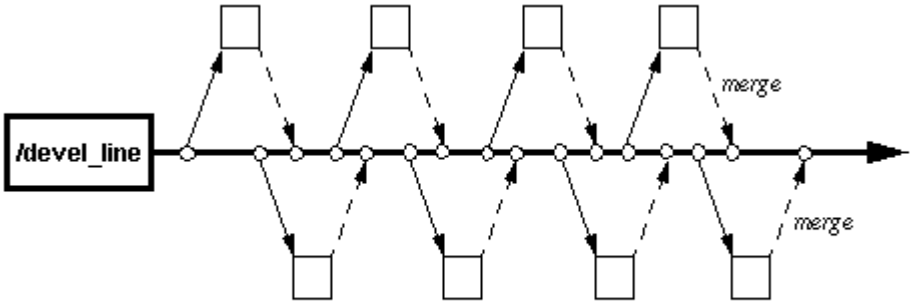


Figure C2a: A separate activity-branch for each development task

An activity-branch is a kind of degenerate a codeline, complete with an owner and a policy. The branch owner is the owner for the particular task or activity. The policy of the single-activity codeline is that it is completed (retired) and merged back to a parent codeline once the first logical change to it has been completed.

Variants

C2.1 ***Branch per Major Task (a.k.a Major tasks off-line, Minor tasks on-line)***
A common variant of *Branch per Task* is to use an activity branch *only* for major tasks. Minor tasks are performed "on-line," without any branching off of the codeline. Such tasks are believed to be "short and sweet" and can usually be completed and checked-in before very many other tasks have had a chance to checkout and checkin any of the same files.

Ostensibly, major tasks are those that would be "long transactions": The state of the codeline will have ample time to change between the time the tasks starts and finishes. Often, a certain minimum threshold of estimated changed lines and/or files is used to discern major tasks from minor tasks. This is an attempt to estimate the inherent complexity of each task and the potential destabilizing effect it may have upon the codeline. The underlying desire is to mitigate and isolate the risk associated with each change, and its impact upon the codeline.

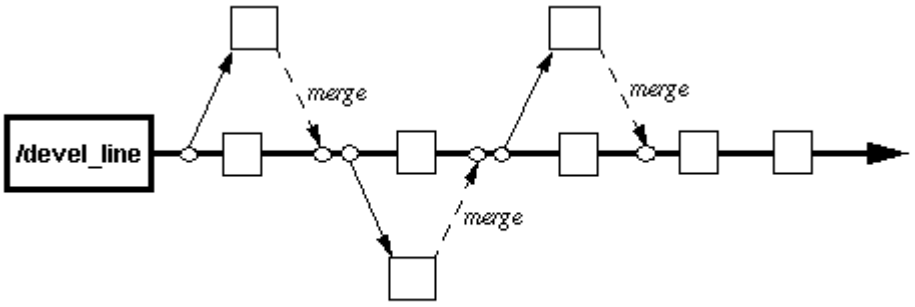


Figure C2.1a: A separate activity-branch only for major development tasks

C2.2 ***Branch per Change-Request (a.k.a. CR-Branch, Change-Set Branch, Change-Package Branch)***

Use a branch to isolate the changes for each change-request (CR). The branch encompasses a single logical unit of functionality (a feature), or a single logical change in functionality (a bug-fix or an enhancement). The same branch should be used for all files that collectively participate in the lone feature or change.

C2.3 ***Personal Activity Branch (a.k.a. Single-User Task Branch)***

A personal-activity branch is simply a special kind of activity branch where the owner of the task is the sole performer (developer) of the task. It may be accompanied by a branch-locking policy to ensure that the personal activity branch is also a private activity branch (see *Private Branch*).

Resulting Context

- The changes along activity branches are isolated from one another so their corresponding modifications do not interfere even when their corresponding file sets overlap.
- Making changes independently separable from each other lessens the burden, and the risk, of integrating them with the myriad other changes in the codeline. Packaging them up as branches makes them easier to identify, trace, and query as a group.
- CR branches and activity branches can be a convenient "packaging" mechanism for grouping together multiple file revisions as a single, logically coherent change. However, when versions from one branch are integrated to another branch, the newly integrated versions lose the grouping provided by the original branch.
- Merging must be performed for changes that didn't even require concurrent modification of the same files. Although the "copy-merge" will be trivial, numerous copy-merges can add up to significant overhead for files that didn't really change. The variant *Branch per Major Task* addresses this by reducing the number of trivial merges at the expense of less isolation for supposedly minor changes.

Related Patterns

Codeline Policy, *Codeline Ownership*, and *MYOC* were mentioned above. If activity branches start to depend on other activity branches (in addition to depending on a baselevel of the codeline), then a *Subproject Line* should probably be used.

This pattern has some interesting similarities with *Thread per Request* [Schmidt96].

C3. Codeline per Release

Pattern ***Codeline per Release***

Context During the course of development you will need to perform several software releases. Work tasks are often organized around such delivery dates and their milestones. You would like the branching structure used to readily accommodate the organization of work tasks performed by developers working toward the common goal of a release (contrast this with a project that may be more architecture-centric than delivery-centric).

Problem How should software releases, release engineering, and development for a particular release be reflected in the branching tree of the project?

Forces

- To be sure, labels will be used to tag stable configurations ready for release. But baseline labels alone are insufficient to help structure workflow for a specific release.
- Major and minor release, as well as patches will often need to have their maintenance and development tasks occur simultaneously, in parallel from one another.
- Merging causes integration overhead, but some amount of integration is essential for verifying and validating that a configuration to be released is suitable for shipment to customers.

Solution For each planned release (major, minor, and patches), use a separate codeline to organize efforts focused on each individual release-effort.

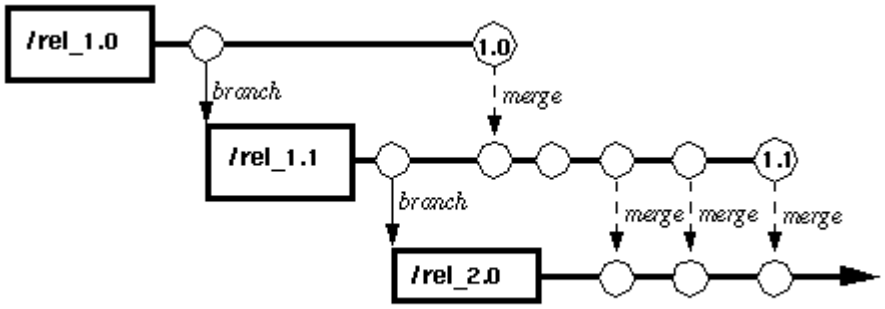


Figure C3a: Codeline per Release for releases 1.0, 1.1, and 2.0

There are essentially two ways of doing this: you can use *Early Branching* the release lines, or *Deferred Branching*. With early branching, you will create a release-line as soon as efforts toward that release begin to take place. With deferred branching, a new release-line is created only when changes specific to that release begin to take place. Changes that need to go into that release as well as in an existing release-line would take place in the previous release-line that also needs the change.

For example, a bug-fix might be needed both for release 1.1 as well as release 2.0. With early branching, that bug-fix would go into both the 1.1 and 2.0 release-lines (possibly creating the 2.0-line if it didn't already exist. With deferred branching, if the 2.0-line didn't exist yet, the bug-fix would go into the 1.1-line only and creation of the 2.0-line would be deferred until efforts began for a fix or feature that was needed *only* for release 2.0 (or later).

Variants

C3.1 ***Codeline per Major Release (a.k.a Major releases off-line, Minor releases on-line)***

In this variant, new release-lines are created only for major releases. So a release-line might be created for release 1.0, and also be used for releases 1.1, 1.1.1, 1.2, etc. (all releases of the form 1.x or 1.x.y).

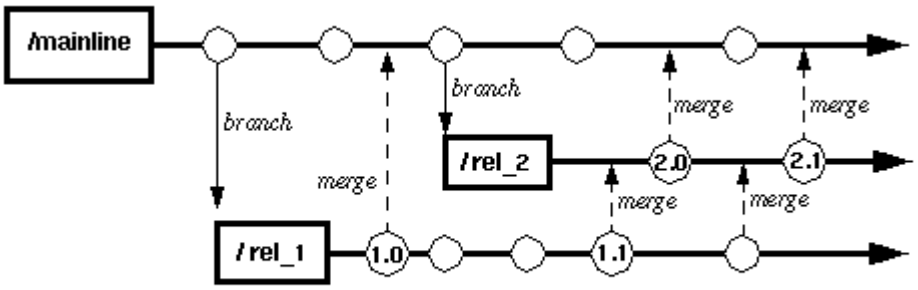


Figure C3.1a: Codeline per Major Release with Mainline

This variant can also be implemented using either early branching, or deferred branching. But since it already uses deferred branching to some extent (at least for releases with the same major version number), it is more commonly implemented with early branching.

However, it may not always be feasible for each minor release and patch effort to happen on the same codeline. Some efforts will need to continue while release engineering is performed on a more stable branch (see *Anti-Freeze Line*). So there will frequently be separate branches for minor-releases and patches, which merge back to the major release-line when finished. But there won't necessarily be a separate branch for *every* such release.

Resulting Context

- The branching structure more closely reflects the organization of work efforts for the various releases of the software.
- Additional integration effort is required, especially for propagating changes needed by multiple release lines from earlier releases to subsequent releases.
- If the propagation hierarchy grows to a depth of three or more, this can impose an extremely noticeable amount of integration overhead. *Deferred Branching* and *Codeline per Major Release* help to mitigate this risk, but not to eliminate it.

Related Patterns *Early Branching, Deferred Branching, Overlapping Release-Lines, Parallel Maintenance/Development Lines, Anti-Freeze Line*

It is extremely useful (and often necessary) to use a *Mainline* when using this pattern, or its variant. Otherwise the project branching tree can get too wide and unwieldy. Both *Codeline per Release* and *Codeline per Major Release* may be used with either the *Stable Receiving-Line* or *LAG Line* variants of mainline (as well as with both of them at the same time).

C4. Subproject Line

Pattern ***Subproject Line***

Aliases *Sub-Codeline, Change-Line, Project Branch, Persistent Branch*

Context You are using *Branch per Task* or *Branch per Major Task*, and you have a somewhat large-grained change-task to perform. It is large enough to split into several smaller change-subtasks, at least one of which is also deserving of its own separate branch. Some

of the subtasks may need to be sequential, with certain subtasks dependent upon changes made in their predecessor tasks. Or perhaps some of the subtasks will be performed in parallel. In either case, you can't permit the individual subtasks to be integrated back into the codeline until all of the subtasks are complete; otherwise the codeline may be in an inconsistent state for other developers working on other codeline tasks in other workspaces.

Problem How do you organize the subtasks of a large-grained task that needs to effect changes to the codeline?

Forces

- We want the composite task to enter the codeline as a single change transaction.
- The composite task really is large enough that it needs to be further subdivided into discrete subtasks.
- Checking-in individual subtask-change back to the codeline may compromise the integrity of the codeline if not all subtasks are finished yet

We could just have later tasks refer to the branches for their predecessor tasks. But this has a few problems of its own:

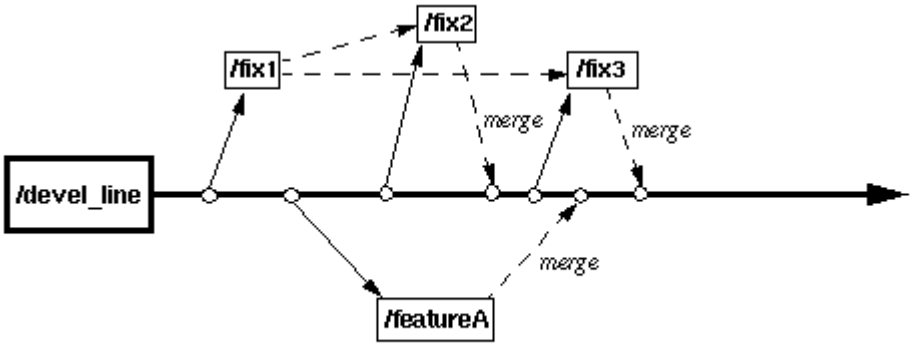


Figure C4a: Multiple dependent activity branches (undesirable)

- Our activity-branch now depends on other branches that aren't yet part of the codeline. This can wreak havoc at integration time if the branches aren't integrated in exactly the right order.
- Each subtask has to remember one more branch-dependency than its predecessor. This can become cumbersome and error prone for three or more subtasks.
- If parallel development is taking place between the subtasks, it may be dangerously easy for undesirable codependencies to arise between their various activity-branches. This can make for an unwieldy integration of all the subtasks back into the original codeline.
- It can make the branches in the version-tree take on a shape that doesn't really reflect the hierarchical nature of the composite task. This can be very misleading to people.

Solution Create a branch for the composite task, and then create subbranches off the composite-task branch for the subtasks (or just the major subtasks). When a subtask is completed, integrate the subtask branch back into the composite-task branch. When all subtasks are completed and have been integrated into the composite-task branch, then integrate the composite-task branch back into the codeline.

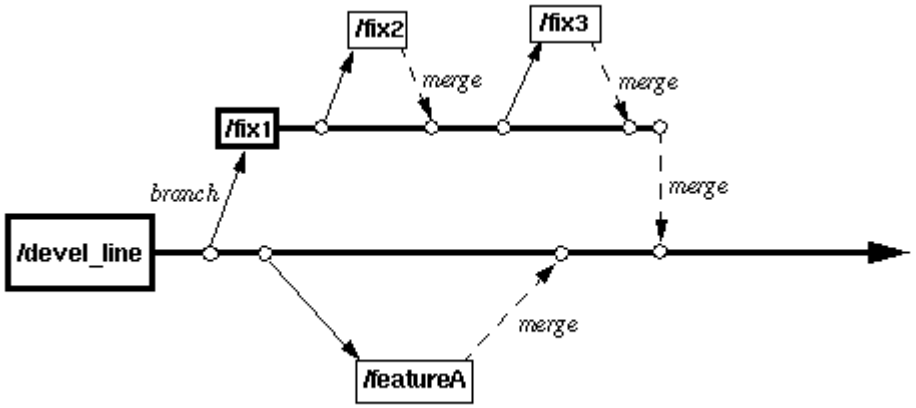


Figure C4b: Using a Subproject Line to decouple branch dependencies

If the composite-task is for a fix, then this is sometimes dubbed a *Fix-Line* (not because it is for fixes in general, but because it contains logical changes toward implementing a single bug-fix). Similarly, if the changes are for implementing a single feature, this is often called a *Feature-Line*.

So we branch off the codeline to create a *Change-Line* or *Subproject-Line*: a line that incrementally receives changes for the particular fix or feature. The change-line should be treated like a codeline, having its own *Codeline Policy* and *Codeline Owner*. When all change increments have been completed and/or merged into the change-line, the change-line may then be merged into its parent codeline.

Variants

C4.1 ***Personal Codeline (a.k.a. User-Branch)***
A single-user-branch or personal codeline may be used when the subproject in question consists entirely of related tasks that are carried out by the same developer. It may simply be a bunch of maintenance tasks for the same release, or perhaps a more significant feature or enhancement. The personal codeline may be accompanied by a branch-locking policy to ensure that the personal codeline is a "private line" (see *Private Codeline*).

C4.2 ***Experimental Line (a.k.a. Prototype Line or Proto-Line)***
A codeline for experimental development. It may be an experimental prototype that is discarded (even upon success) or any other work that isn't yet planned for any known release, and might be totally abandoned.

C4.3 ***Multi-Project Lines (a.k.a. Multi-Product Lines)***
This is basically the same pattern used at a coarser granularity. Whole project-lines or product-lines are used and releases are configured by selecting the appropriate stable baselevel of each project/product-line and integrating them together. This is an extremely advanced practice used by very mature software shops that have developed a sufficient base of separable and reusable components or products in an integrated product family. In essence, each standalone software asset is escalated to full-fledged product/project status (even if they are not provided to customers on an individual basis).

The hard part is balancing the needs of the individual assets against the needs of the integrated product. There will be much pressure to splinter off project lines for specific deliverable products. Organization's that successfully use multi-project lines are able to manage and resist short-term needs to splinter the project-lines for each project and reap the long-term benefit. This is no small organizational feat.

Resulting The composite-task-branch is a mini-codeline or change-line of sorts. It falls short of being a release-line: it still spans multiple change-tasks during its lifetime, but it is shorter-lived

Context and doesn't encompass all the functionality of a release. This is why we refer to it as a *subproject-line*.

Once again, we have "added another level of integration" to solve our problem. In this case, we just recursively expanded our usual strategy of activity-branches on a codeline to another level of scale, in between the level of a single-activity branch, and release-line or what we might otherwise normally think of as a codeline. We created a subcodeline to fill in the gap between the codeline and the activity-branch. The subcodeline is simply a microcosm of its parent codeline.

This lets us employ frequent incremental integration (*Merge Early and Often*) on the subcodeline without compromising the integrity of the parent codeline. The project version tree still reflects the hierarchical nature of the composite task and we aren't forced to introduce any weird dependencies or codependencies between the subtask-branches. We do pay the extra costs associated with merging, but merging for sequential subtasks should be trivial, and merging for parallel subtasks will be no more difficult than if we had never created the subcodeline (and might even be less difficult).

Related Patterns *Branch per Task* (or *Branch per Major Task*), *Merge Early and Often*, *Codeline Policy*, and *Codeline Ownership* are all used for the subproject-line. The subproject-line may itself be viewed as a recursive form of *Branch per Task* or *Branch per Major Task*. It may also be viewed as a kind of *Policy Branch* whereby the subproject-line was created due to the need for a codeline policy that is incompatible with its parent codeline (perhaps with regard to its stability/volatility, or its integration rhythm: the mean time between integrations).

C5. Virtual Codeline

Pattern ***Virtual Codeline***

Aliases *Floating Label Line, Just-In-Time (JIT) Branching, Branch on Demand*

Context You wish to use branches and codelines but one or more of the following is true for your particular situation:

1. Your VC tool does not support symbolically named branches, and you won't be able to obtain one that does any time soon.
2. You would like to "optimize out" some of the extra merging associated with branching (especially for files that don't have any concurrent changes to be reconciled against the codeline)

Problem How can use mnemonic branch names if the VC tool you are stuck with doesn't support them? And how might you eliminate some of the trivial yet frequent merging for files whose contents don't change after they've been merged to the codeline?

Forces

- Branches are good for hierarchically organizing workflow and change-flow into isolated development paths.
- If your VC tool doesn't support the use of meaningful branch names, it can be very difficult to conceptually keep track of which branches are used for what and of the relationships between branches.
- When merging a file from a branch or codeline into another codeline, even if there are no concurrent changes to be reckoned with, the contents of the version still needs to be copied to the receiving codeline. This is called a "*copy-merge*".
- For large efforts, although "*copy-merging*" may be trivial to do, if it needs to be done

for lots of files, there may be non-trivial overhead associated with it. Not only does it require extra time, it may require redundant storage and unnecessary network usage (depending on your VC tool).

- Selecting versions by branch-name can sometimes be less efficient then selecting them by baseline-label. For large projects with lots of files, this can affect the performance of builds, queries, and checkout/checkin operations.

Solution

The majority of VC tools support the concept of a "*label*": a user specified tag attached to one or more revisions. Even if your VC tool doesn't support named branches, there is a good chance it supports labels. Labels are typically used to group together revisions that form a stable configuration of the system or one of its components, or of all the revisions in a logically coherent change-task.

So, implement a "virtual" codeline using a *Floating Label*! A floating label is a label whose name remains the same but whose definition (the set of files and file versions associated with the label) changes periodically. You will still need to use static labels as well (just as you would with branches) to denote named stable baselevels of the codeline. But the floating label will serve the same purpose as a named branch: a mnemonic selector for the latest version of all files on the codeline. Here's how to do it:

1. All revisions that should initially participate in a "codeline" or branch are initially tagged with a label having the desired branch name.
2. To make a change to a virtual codeline, first select an appropriate baselevel on the "codeline" in your workspace. Or select the codeline label itself if using static version selection (if you are using dynamic version selection, you might not want your workspace to be automatically updated with new versions when the floating label definition changes).
3. When a file needs to be checked out, if no one else has it checked-out, and it is the latest revision on its branch, then just checkout the file *without branching*. Otherwise, checkout the file onto a branch. This is called "*Just-In-Time* Branching" or "*JIT Branching*." It may be easiest to write a script or macro that does this automatically.
4. Checkin your completed file changes as you normally would. They won't become part of the "virtual codeline" until they are explicitly tagged with the codeline-label.
5. When the time comes to merge changes back to the virtual codeline, no copy-merging is required. The revisions don't have to come to the codeline, the codeline-label comes to the revisions. Only those files that have genuinely concurrent changes will require merging.
6. After the merge is complete, create a new baselevel label if that is what you would normally do. Then tag all the new revisions in the change with the codeline-label. Files that had no concurrent modifications during the change are labeled as is (since no "*copy-merging*" was necessary), files that required merging will have the codeline-label applied to the newly merged result.

If it is confusing to determine which new revisions to label, you can just re-label the entire "codeline" (but that could be very time consuming for a large project).

Resulting Context

- Allows you to use mnemonic names for your "virtual" codelines and branches.
- Avoids the overhead of unnecessary "*copy-merging*".
- Requires additional effort from codeline owner to explicitly redefine the floating codeline-label, applying it to newly integrated revisions.

- If a single change is large or complex, or if multiple changes are integrated at once, it may be difficult to readily keep track of the new revisions that need to be labeled. Labeling the entire workspace may be undesired or very time consuming for lots of files.
- You can't use "native" (non-virtual) branches to group activities. You'll have to use labels instead if you need that.
- Labels don't accommodate the nice hierarchical branch-path names that some VC tools provide (e.g. ClearCase and Perforce). Furthermore, if your VC tool provides nice visual displays of version trees, you forfeit that benefit as well.
- JIT branching employs file-based "native" branches as a low-level optimization underneath a virtual branch label. For most intents and purposes, the branch label "looks and feels" like a single line of development. But the resulting shape of the native branch-tree may not be fit for human cognition.
- Assumes that your VC tool supports labels, and that it provides facilities to query what the latest file revision on a branch is, and whether or not the file is currently checked-out by others. This is quite common however (more so than named branching).
- If lots of file-based branches take place, many files may possess very wide version trees (especially when files are commonly "checkpointed" before committing a change). If left unchecked, this can begin to suffer from some of the performance issues mentioned above; and full-revision names may actually begin to exceed the maximum allowed command-line length for your platform.
- If the above situation occurs, periodic copy-merges back to a native ancestor branch may be required. But these should be few and far between and can be conveniently scheduled during "low usage" hours.

Related Patterns *Codeline Ownership, Staged Integration Lines*

C6. Remote Development Line

Pattern ***Remote Development Line***

Aliases *Site Development Line*

Context You are collaborating with developers at a geographically remote site. The remote developers might be contractors participating in a subcontract agreement with the master-site. It might also be the case that the remote site is using a mirrored repository instead of centralized repository access. Both the local site and the remote site and performing change-tasks that will ultimately be part of the same codeline. But you want changes at the remote site to be verified or validated by someone at the local site before they become part of the master site's local codeline.

Problem How do you organize remote development tasks so that they don't unduly impact the codeline used by the master-site, while still allowing the remote developers to work without unnecessary delays?

Forces

- If the remote site is participating in a subcontract agreement, verifying their changes before integrating them into the codeline may actually be a contractual obligation.
- If mirrored repositories are used, care must be taken so that conflicting version trees aren't created at each site. For example, if each site does a checkout that creates

version 1.5 of a given file, it will be hard to correctly synchronize the repositories if they each have incompatible ideas of what version 1.5 looks like and who made the changes and when. Although work may still occur in parallel, the resulting changes to the names and structure of the version tree at either site will need to be mutually exclusive from the other.

- You don't want remote changes to compromise the efforts of the local site. Such effort may take even more time than usual to "back out" because of the geographic separation between the two sites (especially if they are in different time zones)
- You don't want the remote site to be "blocked" any more than necessary when their changes are being verified (especially if you are paying a premium for their efforts, even when they are waiting to hear back from you)

Solution

If work on a particular codeline (e.g. a release) is split across multiple sites, then give each remote site its own site-specific codeline while the "main" site uses the primary codeline. The remote sites can continue working on their remote development lines while the master site periodically chooses convenient times to verify remote changes and integrate them back into the primary codeline. The remote sites then resync their remote lines with the primary codeline at appropriate intervals. or they may simply choose to always branch off the primary codeline onto their remote-line for each checkout.

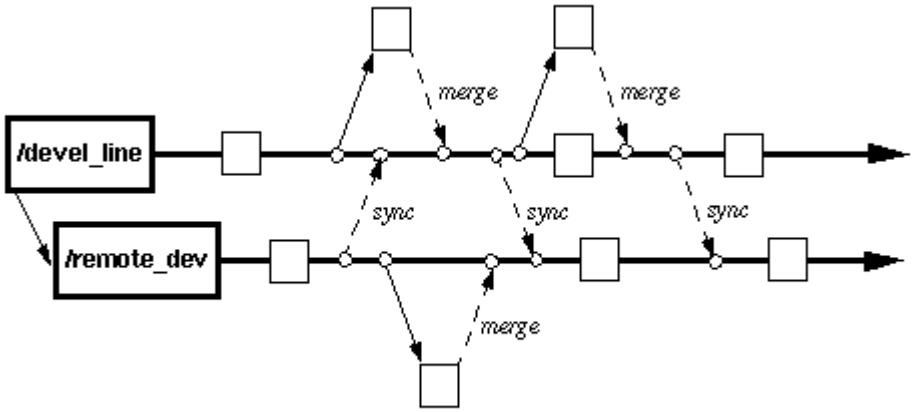


Figure C6a: Remote Development line for local and remote development sites

An additional possibility would be to have the master-site also use a separate (non-primary) development codeline, and to have all sites (master and remote) integrate to the same primary codeline, under the supervision (or strict control) of the master site. In this case, all sites keep their development lines in sync with the master integration line.

Resulting Context

The extra level of integration adds some additional merge overhead, but additional verification/control effort was already required, so some of that would have been necessary anyway.

Since the remote site is a separate entity there is decreased communication between developers at disparate sites. Integration and synchronization is in fact a form of communication between branches and codelines and it too will need to reflect this difference in communication patterns between people at the same site and people across remote sites.

Using a site-release codeline allows the remote site to accumulate changes in the site codeline, which may then "batch up" those changes and allow the master site to incorporate them at their own pace (like a remote *Docking Line*).

Often, there will be instances of *Branch Dictatorship*, between the remote sites. Where the remote site is not permitted to create file versions on the primary site's codeline, but must instead create them on the remote site's codeline. This *Branch Mastership* is little more than a form of *Codeline Policy* specifically designed to enforce a strict notion of *Codeline Ownership* for codelines at the primary site, and at each remote development site.

Related Patterns *Codeline Policy, Codeline Ownership, Docking Line, Restricted-Access Line, Branch Dictatorship*

C7. Component Line

Pattern ***Component Line***

Aliases *Module Line, Subsystem Line, Product Line*

Context *Code Ownership* is being used to ensure accountability and responsibility for related elements of a configuration component (files, modules, subsystems, etc.). After some time however, it becomes clear that parallel development of the component is necessary. Several fixes may need to happen at the same time, or may need to occur while other feature changes are in progress. The code owner is simply incapable of making all these changes by himself within the requisite time period.

Problem How do you maintain the benefits of code-ownership when you need to allow people other than the code-owner to modify the owner's code?

Forces

- If everybody is responsible for something, no one is responsible. You still want a single owner to be accountable for the code.
- If they can't modify the owner's code, important features and fixes won't be completed on time and may hold-up a patch or release.
- Having developers wait for official approval from the code-owner before checking-in changes may still be an unacceptable or unnecessary time delay, especially for critical-path tasks.

Solution Create a new codeline reserved exclusively for development and maintenance of the code-owner's source code. Make the code-owner be the codeline-owner for the components under development on this particular codeline. The code-owner sets the policy for the codeline, deciding when and if changes by others may be made "on-line" or on a separate change-activity branch. The rest of the project will use only stable baselevels of the component-line for integration into releases, or else the code-owner will be responsible for propagating changes from the component-lines into the overall product-line or main development line.

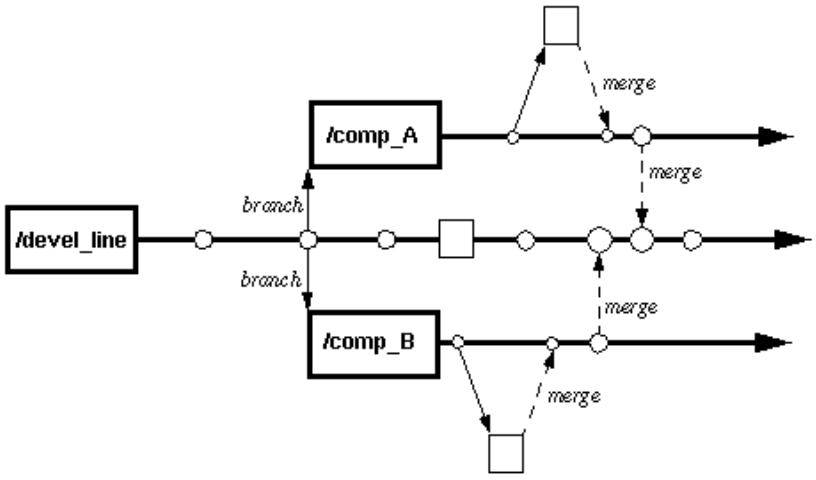


Figure C7a: Two Component Lines that merge back to the development line

Variants

C7.1 ***Multi-Product Lines (a.k.a. Multi-Project Lines)*** This is the same as pattern variant C4.3, *Multi-Project Lines*.

Resulting Context

- Changes to the owner's code may be made in parallel using multiple developers. This reduces development time for completing changes to the particular component.
- The added level of integration gives the code-owner the final say over what goes into the production version of the component under development on his component-line.
- Additional time and effort is required by the code-owner to verify, and perhaps even perform, merges into the component-line. However, this effort does not significantly impact others working on the component-line.
- The additional integration effort may impact the efforts of those waiting for stable versions of the component-line to be included/integrated into the product-line, but no more than they would be if the code-owner had to perform all changes himself, or if developers had to wait for approval from the codeline owner to checkin their changes.

Related Patterns

The code-owner will need to be the *Codeline Owner* for the component-line and adopt an appropriate *Codeline Policy*. The code-owner will also need to choose from branching and merging patterns such as *Branch per Task*, *Branch per Major Task*, *MYOC*, *Docking Line*. If a component-line is used in conjunction with a docking-line, or if the component-line itself must be periodically merged into a main development line, then this is an example of *Staged Integration Lines*.

C8. Platform Line

Pattern ***Platform Line***

Aliases *Variant Line, Permanent Variant Branch, Platform Branch*

Context You are developing software that will need to execute (and perhaps even build) on multiple platforms or environments. The need to work in the face of varying operating environments may be a new requirement, or an initial one. You have made an honest attempt to handle platform build and execution discrepancies using environment-specific directories and files. However, the nature of this particular project's multi-platform requirements has proven this approach to be unsatisfactory.

Problem How should platform-specific build and execution information be identified, tracked, and controlled for a multi-platform software project?

Forces

- Normally, platform-specific configuration elements (files and directories) with common interfaces and organization are the most appropriate solution. But in this case, this approach has already proved unwieldy.
- Build and run-time information must be easily separable and discernible for each platform.
- Platform-specific changes should be able to occur independently from one another without impacting any other platform.
- There may exist the need to build or execute for more than one platform simultaneously (while the software is already building or running).

Solution For a new platform requirement where no multi-platform needs previously existed, create a new codeline for that platform (a *platform-line*). If multiple platforms are needed, then a new codeline for each platform. All creation and modification of platform-specific content should take place on the codeline for the given platform. Files that require no special awareness of differing platform requirements can remain on the main (common platform) development line. The platform line should never merge back to the development line while differences in operating platforms are needed in source code or build files.

Initially, it may be sufficient to create only a single platform-line for the first additional platform.

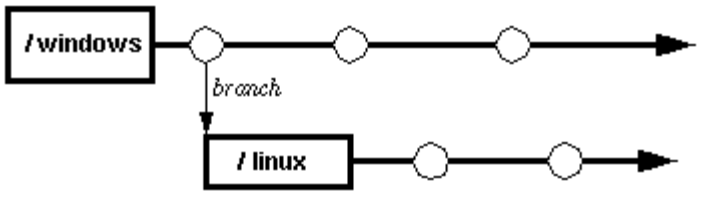


Figure C8a: Permanently diverging codeline for a new OS platform

But, if multiple-platforms are being used, you will probably need to go through the existing codebase and separate out platform differences into two or more codelines (one for each platform) and maintain a separate primary codeline for platform independent-code.

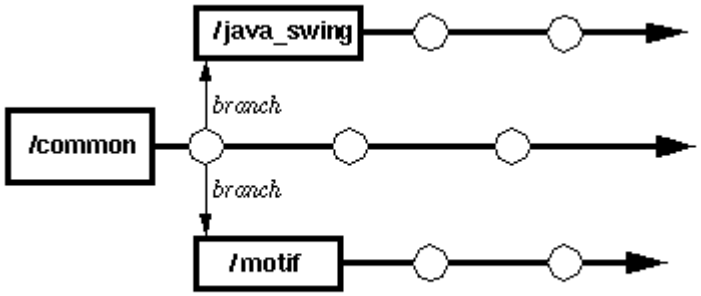


Figure C8b: Platform lines sharing a common base subset

In more complex cases, there may be platform families: subsets of platforms with common needs and constraints. In these situations, you may need to create platform-family-lines, in addition to specific platform-lines and a lone common-platform line.

For example, you might have a program that needs to run on multiple varieties of Unix (Solaris, AIX, HP-UX, Linux) and Windows (Windows'95 and Windows/NT). The common-platform line will be the primary development line and contain platform independent files and changes. Branched off of that will be the platform-family lines (e.g., Unix and Windows) containing files and changes common to all platforms in the family. Finally, branching off from the platform-family lines will be the lone platform lines (e.g., Linux, AIX, Windows'95, etc.) for files and changes specific to that platform alone.

Resulting Context

- Platform-specific files, their revisions and configurations can be easily selected by using the platform-line (and platform-family line if present) for a given platform.
- Once the desired platform lines have been selected in the workspace, the correct contents and information should be present to properly build and execute the software for the particular platform-variant.
- Examining the branch to which a file revision belongs can easily identify platform-specific files and changes.
- The use and selection of platform lines make only one platform visible at a time in the developer's workspace. This means less confusion about which platform a set of files or changes is intended for.
- You won't be able to build for multiple platforms at once. Building for multiple platforms simultaneously will require not only a separate build, but the workspace will first have to be reconfigured for each platform. This is one of the main reasons why using platform-directories is preferable if it can be done.
- Most version control tools let you select specific revisions of files, but not of directories. They will give you all files in a directory, even if some of those files aren't needed for a given platform. For tools like ClearCase that do version control the contents of directories, branching of directory-contents is more complex than of file-contents, and should not be taken lightly. This is another reason why platform-directories are preferable to platform-lines when possible.

Related Patterns

Codelines for components that exist only for a certain platform, or platform family, can be treated like a *Component-Line*. The use of *Codeline-ownership* is especially important for platform lines since they rarely converge back to their parent codelines. The codeline/platform owner will have to be on the lookout for files that can be merged back to a common platform line or family, as well as for file revisions that need to move from a common platform to a family platform, and from a family platform to a single platform. The *Codeline Policy* used for platform lines will need to make special mention of when and why to merge to and from platform lines, platform-family lines, and the common platform line.

Branch Structuring Patterns

- S1. *Mainline*
- S2. *Parallel Maintenance/Development*
- S3. *Overlapping Releases*
- S4. *Docking Line*
- S5. *Staged Integration Lines*
- S6. *Change Propagation Queues*
- S7. *Third Party Codeline*
- S8. *Inside/Outside Lines*

S1. Mainline

Pattern ***Mainline***

Aliases *Main Trunk, Main Anchor Line, Home Line, Ground Line*

Context During the development and maintenance cycles, multiple codelines are created for various reasons. Typical codelines are release-lines and maintenance-lines and integration-lines. This is particularly true when using *Codeline per Release*, *Parallel Maintenance/Development Lines*, and *Overlapping Release Lines* (or any of their variants). As time goes on, codelines continue to cascade off of other codelines making the project version tree become wider and wider.

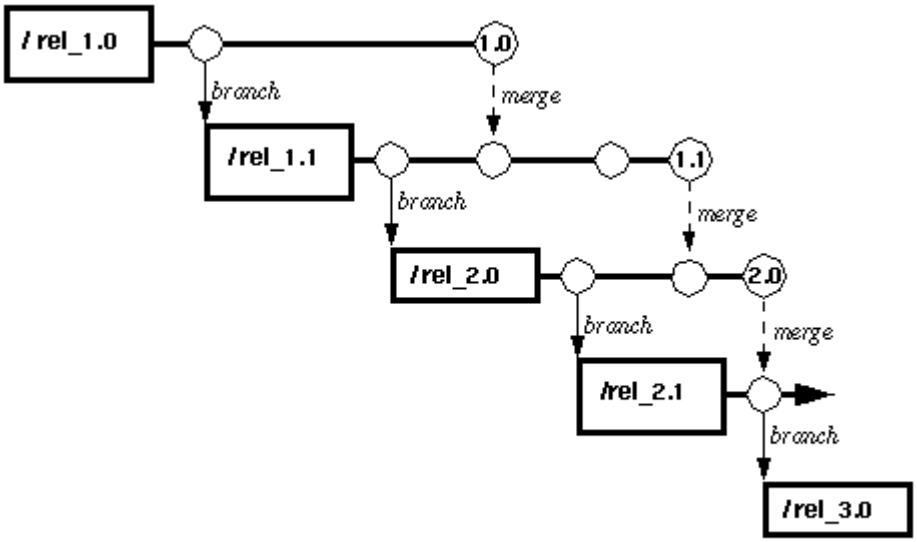


Figure S1a: Continually Cascading Codelines (undesirable)

Problem How do you keep the number of currently active codelines to a manageable set, and avoid growing the project's version tree too wide, and too dense?

Forces

- Every codeline that is created typically requires a merge back to its parent branch at some point in time. So more codelines means more merging, and more merging means more synchronization effort.
- It seems only natural to branch each new release-line off of the codeline for its immediately preceding release.

Solution Rather than continually cascading branches upon branches creating an extremely wide and unwieldy version branch-tree (requiring an enormous amount of synchronization between each parent-child branch pair), keep a "home branch" or codeline at the trunk (or right next

to the trunk) of the branch tree.

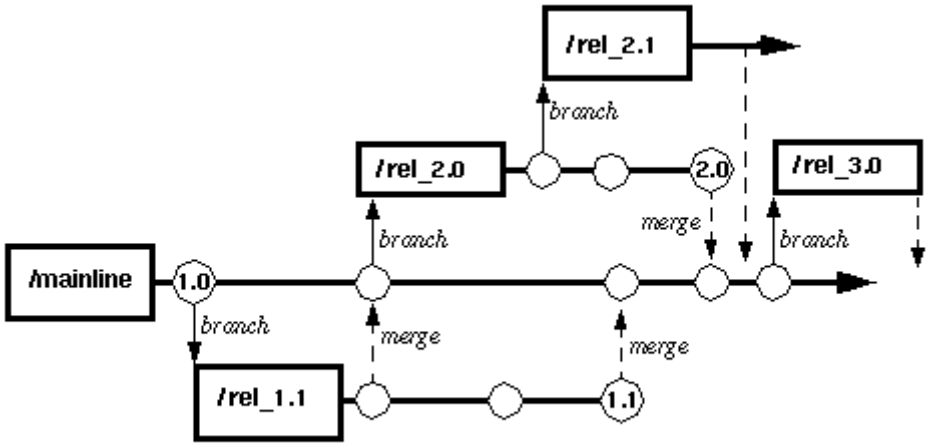


Figure S1b: Cascading Codelines with a Mainline

When the time comes to create a codeline for a new major release, instead of branching the new release-line off of the previous release-line, merge the previous release-line back to the mainline branch and branch off the new release-line from there.

The process of merging back to mainline for a particular codeline or branch is sometimes referred to as "*mainlining*," "*trunking*," "*homing*," "*anchoring*," or "*grounding*."

Variants

S1.1 ***Stable Receiving-Line (a.k.a Stable Mainline, Main Integration Line, Base Integration Line)***

Keep a stable, reliable main development trunk that is used solely for importing (receiving) stable bases from other codelines. No development work ever takes place directly on this codeline, and all integrated changes must come from some other codeline (not a single discrete activity branch). The only exception to this rule is that integration changes may be performed for ensuring that the codeline builds and functions consistently.

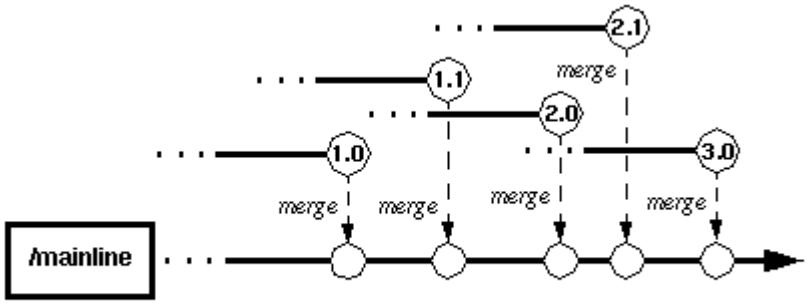


Figure S1.1a: A Stable Mainline for receiving stable baselines only

S1.2 ***LAG Development Line (a.k.a. Main Development Line, Central Line, Main Stream)***

Use the trunk as the latest-and-greatest (LAG) development line which evolves forever and to which all codelines for previous releases are eventually merged after they have been "retired." The trunk gets used as the development line for the next/latest development

release (not maintenance, but development, as in significant enhancements and new features). Thus, when work for release B2 is ready to begin and work for release A1 is completed or is tapering off, then a new branch is spawned to finish up the A1 effort (see *Deferred Branching*) and the LAG-Line is used for work toward release B2 (which is the new "latest and greatest" development effort).

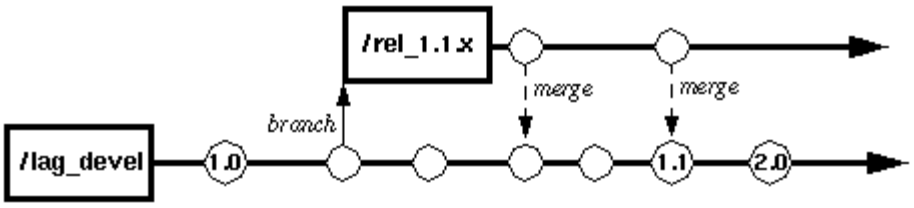


Figure S1.2a: LAG Development Mainline

The process of doing merging for a particular codeline or branch to the LAG-development line is sometimes referred to as "*LAGging*," "*mainlining*," "*LAG-lining*," or "*mainstreaming*."

Although often used as a *Mainline*, a LAG-Line may also be used in conjunction with a *Stable Receiving-Line* as the mainline:

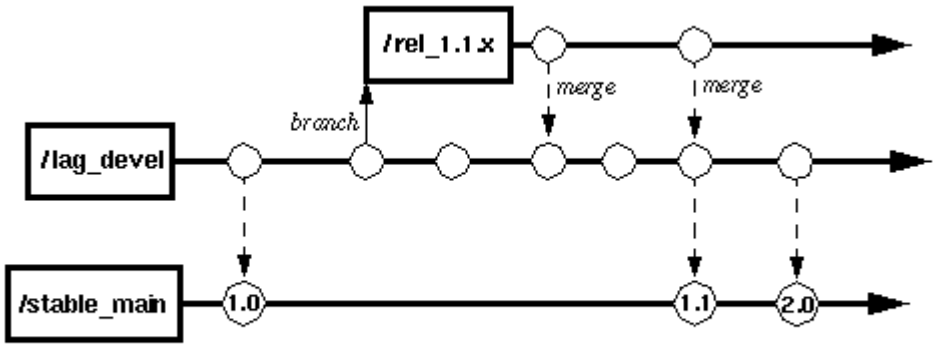


Figure S1.2b: LAG Development Mainline hooked up to a Stable Mainline for receiving releases

Resulting Context

- Reduces merging and synchronization effort by requiring fewer transitive change propagations.
- Keeps full revision names to a manageable length (both conceptually, and physically - so as not to approach the maximum limits of the length of the command-line).
- Provides closure (closing the loop) by bringing changes back to the overall workstream instead of leaving them splintered and fragmented.

Related Patterns

Codeline per Release, Parallel Maintenance/Development Lines, Overlapping Release Lines, Early Branching, Deferred Branching

Also, the last promotion-level in a set of *Staged Integration Lines* often serves double duty as a stable mainline.

S2. Parallel Maintenance/Development Lines

Pattern ***Parallel Maintenance/Development Lines***

Aliases *Parallel Fix/Feature Lines*

Context You've finished a release of a version of the project and need to start, or continue development on the next major release.

Problem How do you conduct development of the next major release while at the same time responding in a timely manner to all the many bug reports and enhancement requests that are inevitably going to be logged against the current release?

Forces

- You need to make steady progress toward implementing the new functionality slated for the next major release.
- You need to respond quickly to bugs and enhancements logged against the current release.
- Critical bug-fixes and enhancements need to be effected immediately, often well before the next major release is ready to ship.
- Maintenance effort (bug-fixes and enhancements) in the current release may be incompatible with some of the functionality or refactoring already implemented in the next release.

Solution Rather than trying to accommodate maintenance of the current release and development of the next release in the same codeline, split maintenance and development off into separate codelines. All bug-fixes and enhancements to the current release take place in the maintenance line, effort for the next major release takes place in the development line. Ensure that changes in the maintenance line are propagated to the corresponding development line in a regular fashion (see *Merge Early and Often*, and *Change Propagation Queues*). One way of doing this is to create two new branches: one for maintenance and one for development.

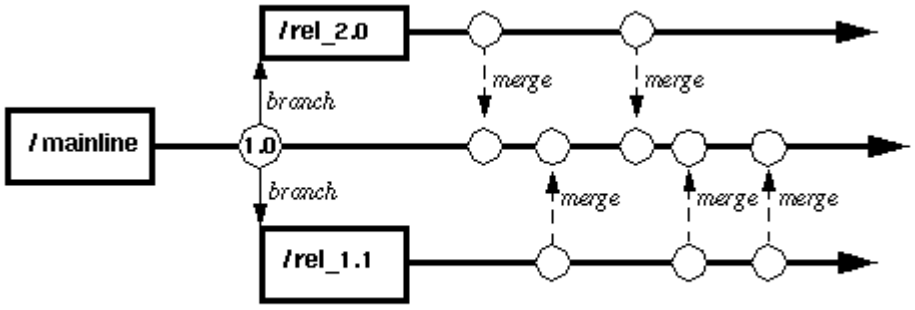


Figure S2a: Parallel Maintenance/Development using two new lines

The more common approach is to create one new branch. If the new branch that is created is used for development, then maintenance work happens on the same codeline is initial development for its corresponding release. So the branch "invariant" is the major release functionality. Another approach uses *Deferred Branching* and *LAG Development Line* by keeping the development on the same codeline and branching off to create the maintenance-line (so the branch invariant is "maintenance" effort versus "new development" effort).

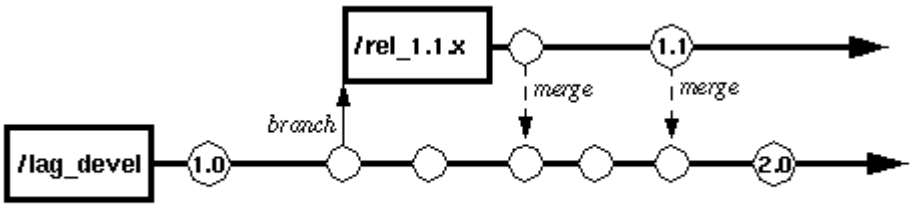


Figure S2b: Parallel Maintenance/Development with a LAG Development Line

Variants

S2.1 ***Parallel Releasing/Development Lines (a.k.a Anti-Freeze Line)***

With this approach, instead of branching off immediately after release, you branch off *before* the release. This allows to *branch instead of freeze*! Instead of freezing the codeline during release engineering activities, a separate line is created for release integration and engineering while allowing other development to continue taking place on the development line (which is why it is sometimes called an *anti-freeze line*). Upon successful release, the release-engineering line becomes a release-maintenance line. It still serves the same purpose of "sync and stabilize" but now it is an ongoing effort that continues even after the release.

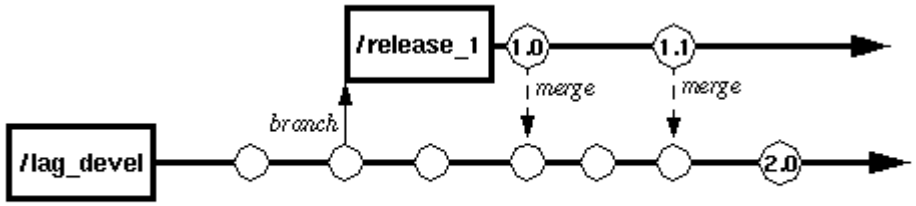


Figure S2.1a: Parallel Releasing/Development using Deferred Branching from a LAG Development Line

Resulting Context

Changes can take place in each of the two codelines at the appropriate pace. Critical fixes and enhancements can be implemented and delivered without immediately impeding future development. Maintenance releases or "patches" can be periodically released without severely impacting development on the next release. The *Codeline Owner* of the development line can set a policy for how and when changes are propagated from the maintenance line to the development.

Related Patterns

Merge Early and Often, and *Propagate Early and Often* should be used to ensure that fixes and enhancements effected in the maintenance line are eventually migrated to the development-line (so the same problems don't reappear in a later release). *Change Propagation Queues* may be used to ensure that changes are propagated in the correct order.

This pattern can be viewed as one particular realization of a *Policy Branch*. The fact is different policies are needed for the maintenance of the old line and development of the new line. Changes on the old line need to be turned around very quickly and should be minimal in their scope. Changes in the development line typically have broader architectural implication, impacting more of the project at once and requiring effort for design, implementation, and testing. The integration "rhythms" for the two codelines are thus drastically different (the maintenance line needs to "salsa" while the development line needs to "waltz").

S3. Overlapping Release Lines

Pattern ***Overlapping Release Lines***

Aliases *Parallel Feature-Lines, Incremental/Evolutionary Delivery Lines*

Context You need to develop functionality for two major incremental releases of the software within a short time span of each other. The schedule for delivering each increment is fairly aggressive (as it always is).

Problem How do you make progress on both development increments without having either one severely impact the other?

Forces

- Cycle Time. It would be nice to avoid any unnecessary waiting or delays for work on the next release.
- Stability. When performing release integration and engineering for the current release, it is vital that the configuration for the release is reliably correct and consistent. Development and maintenance activities on the codeline could easily compromise the integrity of such a "codeline in waiting."

Solution Branch off the current development increment's codeline and start a new codeline for the next development increment. Development on the two codelines progresses concurrently on separate streams of development. From time to time, features and changes in the codeline for the earlier release will need to migrate to the codeline for later release. Propagate these changes early and often at appropriate stable baselevels.

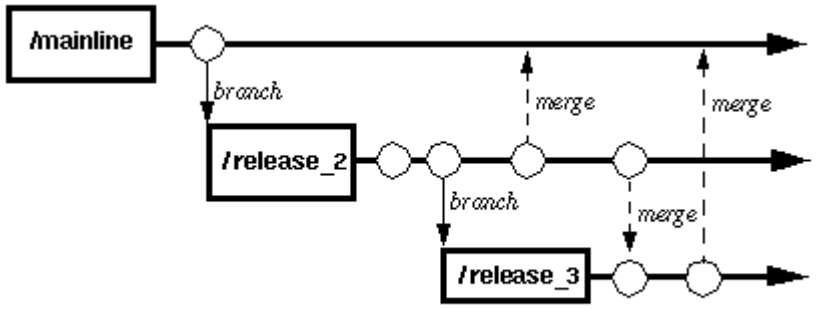


Figure S3a: Overlapping Release Lines with a Mainline

When creating a separate codeline for each release cycle, often, the release codeline will be created as soon as effort begins for the new release. However, if you don't require isolation of the entire effort, *Deferred Branching* may be employed to create the release-line only its development needs to occur in parallel with that of subsequent releases.

Resulting Context Changes can proceed in parallel on the two codelines while in isolation from one another. The current development increment does not need to be slowed down by efforts toward a later incremental release. However, there is a now a dependency of the later release-line upon changes and fixes in the earlier release-line. The integration efforts for these propagated changes are a necessary trade-off over freezing all efforts for release-line while it waits for completion of the other.

Related Patterns As with *Parallel Maintenance/Development Lines*, *Merge Early and Often*, and *Propagate Early and Often* should be used to ensure that changes in the codeline of the earlier development increment are eventually migrated to the subsequent development increment.

Change Propagation Queues may be used to ensure that changes are propagated in the correct order.

This pattern is also a particular realization of a *Policy Branch*. The two development increments have different long and short-term goals and require a different tempo for their development and integration efforts.

S4. Docking Line

Pattern ***Docking Line***

Aliases *Holding Line, Lay-Away Line, Shared Integration Line*

Context The context is similar to that of *MYOC (Merge Your Own Changes)*. A developer has finished a change-task and is ready to merge it back into the codeline. However, in this case the nature of the codeline requires a very paranoid codeline-owner (*Codeline Ownership*) This is usually because the codeline in question is associated with sufficiently high-risk or high-complexity development, or because it demands a greater degree of reliability and consistency than other codelines. Whichever the case, the consistency and integrity of the codeline is relied upon by important people and important tasks, even more than usual.

Problem Who should perform the merge, and who assumes the burden of ensuring it is integrated correctly?

Forces

- As with *MYOC*, we have the case of *dueling ownerships*: the change-owner versus the code-owner versus the codeline-owner.
- *MYOC* also tells us that *follow through* needed to avoid social disconnection (throwing it over the wall), and impact ignorance (lack of awareness of the downstream effects of one's changes). This suggests the change-owner or code-owner should perform the merge.
- The high-risk and/or high-reliability nature of this particular codeline suggests it really *must* be the codeline owner who performs the merge to ensure and preserve the integrity of the codeline (because many critical tasks depend upon it).

All the other forces mentioned in *MYOC* still apply.

Solution Add another level of indirection by adding another line of integration to balance the tension between ownership, follow-through, and reliability. The integration-line is a persistent branch off from the development-line that serves as a "docking area" for changes that need to be integrated back into the original codeline. The change-owner merges (pushes) the changes to the docking-line, integrating with all the predecessor changes. The codeline-owner imports (pulls) one or more changes at a time from the docking-line and integrates them back into the original codeline.

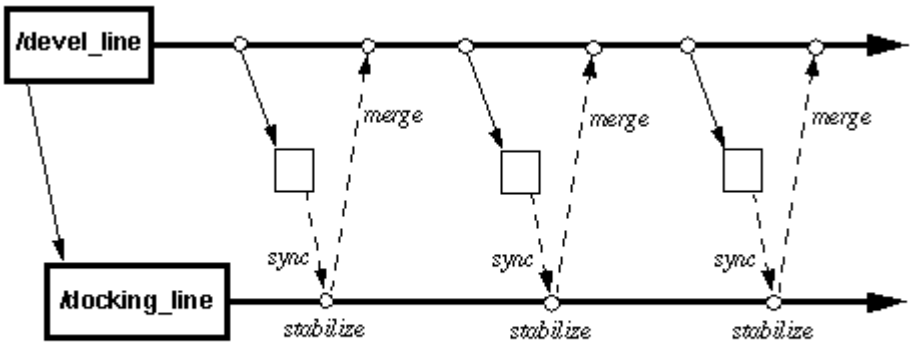


Figure S4a: A Docking line syncs & stabilizes tasks before merging back

Resulting Context

The new integration-line is a "mediator" of sorts between the various owners. The change-owner or code-owner still gets to follow-through by merging their changes to the docking line and having to understand and resolve any conflicts it may have with previous change-tasks. The codeline-owner still gets to ensure the reliability and integrity of the codeline by "syncing" the docking-line with the original codeline and having complete control and authority over its consistency.

The cost of meeting the various owners' competing needs of follow-through and reliability is the added integration effort. However, most of the merging difficulty is handled during the docking-line merge by the person most familiar with the code that was changed. The codeline owner primarily verifies the reliability of the state of the docking line and then merges it back to the original codeline. Since that is where the changes originated, and no other development should be taking place on the original codeline, most of these merges should be trivial because there are no concurrent changes between the docking-line and the original codeline.

If the merged changes in the docking line fail the codeline owner's codeline consistency and integrity checks, then the codeline owner works with the change-owner or code-owner to resolve the issue; but this should be the exception rather than the rule. In the meantime, the person merging the changes to the docking-line avoids having to wait for verification of their merge-efforts before continuing on to another development task. Since the docking-line is separate from the codeline, their work has been effectively isolated, which allows them to return to other important tasks.

Related Patterns

Merge Early and Often should be used to merge changes from the original codeline into the docking-line. Ideally, this is also the case when merging from the docking-line back to the codeline. But if the codeline is high volume or high-load then it may be infeasible to merge change-task at a time. In this case use *Multi-Merge Early and Often*, allowing the docking-line to accumulate a handful of changes before merging it back to the original codeline.

A situation to that of *Docking-Line* might arise if the change-task owner is different from the code-owner of the modified files. This might result in the need for another docking-line (the fewer the better), in which case you have now graduated from a single docking-line to *Staged Integration* lines.

Otherwise you may use only one docking-line and resolve who merges to it (the change-owner or the code-owner). In this case, *MYOC* applies and you can treat the code-owner as the codeline-owner for a *Component Line*.

S5. Staged Integration Lines

Pattern ***Staged Integration Lines***

Aliases *Promotion Branches, Promotion Lines*

Context Development tasks and software changes are frequently required to progress through defined, discrete stages of maturity: Changes are proposed, then analyzed and performed; then perhaps they are promoted through levels of review, debug, unit-test, integration test, and system test.

Other times changes may need to progress through multiple levels of ownership, responsibility, and accountability: Changes might first be effected to a particular file or module, then integrated and tested at the component-level, then the subsystem-level, then the site-level, then the product or system-level.

In either case, there is often the need (or contractual obligation) to provide strict traceability and auditing of changes as they are promoted through these stages of their lifecycle.

Problem How do you use the facilities of your VC tool to track and control changes through defined levels of promotion?

Forces

- Some promotion levels may correspond to the completion of certain events: review, unit-test, system-test, etc.
- Other levels may correspond to states of quality assurance and control to represent some degree of reliability and security.
- Still other levels may correspond to assignment of ownership, responsibility, accountability, and scope (e.g. module, component, or product).
- Not all of these kinds of promotion levels are well suited to the same set of mechanisms provided by your VC tools.
- Among the various promotion levels, some will implicitly require integration to verify/validate correctness, consistency, or to ensure appropriate transferal of responsibility.

Solution Some VC tools and SCM systems already support a notion of promotion levels for tracking and control. If your tool is one of them, then look closely at the documentation to ensure the supported promotion-model is a good match for the model you need to implement. Look at any built-in access controls and transition rules, and how the promotion levels themselves are defined, ordered, and represented. Are they represented as attributes, branches, labels, or something else? How well does that meet your needs?

Otherwise, if the set of promotion levels needed consists predominantly of integration points and the transfer of responsibilities between roles, then use a separate integration-line for each promotion level in the hierarchy.

When original changes are made, they are considered to be at level '0' (often corresponding to a development line or activity-branch). When the change is completed, it is then propagated (merged) to the first promotion-line in the hierarchy. After it is successfully integrated and verified there, then it is propagated to the next promotion-line, and so forth until it proceeds through the final promotion level. The last line in the chain might be a *Stable Receiving-Line*, or it might be development line (like a *Docking Line* split across into multiple lines, but which eventually feeds back to the primary development line).

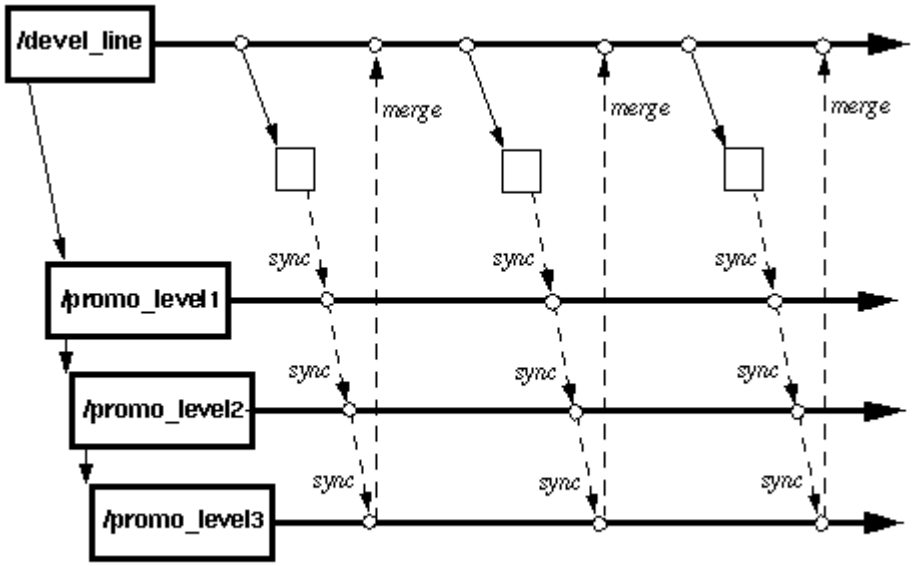


Figure S5a: Staged Integration Lines to represent Promotion Levels

The promotion-level associated with a given revision may be determined by identifying the promotion branch that it belongs to. The progression between levels may be traced by using the VC tool to produce a report of the ancestry of the particular revision (a feature commonly provided by VC tools).

The promotion level of an entire *change-package* (a logically related group of revisions that were modified and submitted together) is harder to determine. Identify the revisions in the change-package and find their branches. They should all be on the same branch (if they aren't, you have detected a consistency error in execution of the change-control process). Then look at the descendents of each revision to ensure the progress through the promotion branches together, and in the proper order. If your VC tool doesn't do this for you in 1-2 simple steps, create a script or macro that does this using the tool's querying and reporting capabilities.

Resulting Context

- Promotion levels for revisions and change-packages can be readily identified and traces.
- The staged promotion branches function like a progression of gates, or a sieve, for transferring and filtering the responsibility and reliability/stability of changes as they progress through the promotion levels.
- Branches are well suited for promotion levels that represent transfer of ownership, responsibility, and accountability. When a change is merged from one branch to the next, the revisions on the new branch are now the responsibility of its codeline-owner.
- Branches also work well for promotion levels that inherently correspond to integration points in the lifecycle. Merging the change from branch to branch performs the necessary integration. However, using Virtual Codelines as the promotion branches can effectively minimize the merge overhead.
- Branches are less than ideal for promotion levels that correspond to static states of quality, maturity, or reliability. Labels and attributes are better suited for this purpose. Branches are inherently dynamic and evolving entities, whereas these kinds of promotion states represent frozen points in time, or static levels of "goodness."

- Another problem with using branches for static "goodness" states: Oft times there will be errors where a change or revision is improperly promoted, or was properly promoted, but later turned out to be flawed. Backing-out revisions from a branch is substantially more effort and more complexity than removing a label or attribute corresponding to the promotion level. Although this is easier to do with a "virtual" promotion-line, it's still more likely to be accomplished with a promotion attribute, or label (as opposed to a codeline label trying to fit two purposes at once).

Related Patterns

In many respects, this pattern is merely a composite of one or more patterns that add an extra level of integration: One can regard a *Docking Line* as a minimal example of staged integration with two lines; so can *Inside/Outside Lines*, and *Branch per Task*.

One or more *Component Lines*, *Remote Lines*, *Subproject Lines*, and *Stable Receiving Lines* will frequently participate in a set of staged integration lines.

For promotion hierarchies that are about three or more levels deep, it may be highly desirable to use a *Virtual Codeline* for one or more of the promotion lines. This will help reduce the amount of trivial "*copy-merging*", which can become substantial after three or more levels of integration!

S6. Change Propagation Queues

Pattern ***Change Propagation Queues***

Aliases *Change Migration Queues, Change Transfer Queues*

Context You have been using *Parallel Maintenance/Development Lines* and/*or Overlapping Release Lines* and you need to propagate changes from one codeline to the other.

Problem How do you propagate changes between sibling codelines in a consistent fashion and in the proper order?

Forces

- You typically want changes to be integrated as early as is conveniently possible, but with minimal impact to others working in the codeline.
- If changes are propagated in a sequence different from the order in which they were effected in their originating codeline, integration may become very difficult or confusing, or may result in more changes than desired being integrated at once (in the case of dependent changes).
- It would be nice if changes could just always be propagated immediately by the person who made them. But this isn't always feasible, depending on the current state of the work in the receiving codeline, or the mismatch between integration intervals of the sending and receiving codelines.

Solution When the number of changes being *Propagated Early and Often* makes it hard to easily remember the completion order and dependencies between propagated changes, then implement an *incoming change propagation queue* for the codeline. Depending upon the SCM tools available to you, this might be done a number of different ways.

1. If all changes to be propagated are tracked in a change control system, create individual change-requests for the propagation of each change. Record its dependency upon the unpropagated-change and the target codeline of the propagated change. Now you can determine the order in which to propagate the changes by looking at the completion times of the corresponding pre-propagation changes. Program a script or macro to query the tracking system for a given codeline and report the propagation

order for all pending changes to be propagated.

2. If this is beyond the capabilities or current usage of your tracking system, see if your VC tool supports *change-tasks* (sometimes called *change-packages* or *change-sets*). A number of VC tools do this, or can be easily customized to do this. If yours does, then see if it tracks change-tasks or provides some kind of completion log to look them up and find out when they occurred, and what their contents are. If so, you can usually customize this in a manner similar to the way we customized the tracking system above.

3. You can write a wrapper script (or a "trigger" if your VC tool provides them for change-package operations) to "submit" revisions in a change-package against a target codeline. The "submit" script can consult the codeline policy for the target codeline and see which codelines are on the export list to receive propagated changes.

4. Once it does this, the ‘submit’ script can use a regular file, or even a database of some kind, to maintain a "propagation queue" for each codeline. When a change is submitted, an "entry" is placed at the end of the propagation queue for all codelines in the export list of the codeline receiving the submitted change. A change may then be propagated into a codeline if and only if it is at the head of the propagation queue, or the queue is empty. After it has been propagated, it is removed from the head of the queue.

Often, the most convenient implementation will be some combination of the above. A script may be used to integrate change-packages with the tracking system to automate the creation, modification, and control of change propagation queues.

Variants

S6.1 ***Auto-Propagation and Queuing***

This goes a step further than the above and usually requires a home-grown "submit" wrapper or trigger script. When a change is submitted, the submit script will try to auto-propagate (auto-merge) the change into the target codelines if their propagation queues are empty. If the queue is not empty, or the propagation cannot be performed automatically without human assistance, then the propagation-task is appended to the codeline's propagation queue.

If auto-propagation succeeds, then the process continues and determines if the change can be auto-propagated to the next codeline in the propagation chain (if there is one), queuing up the change if it can't be auto-propagated.

The relative ease and risk of the auto-propagation attempt depends largely on the sophistication of the SCM tool's merge facilities. It may let the propagation proceed if the tools think the merge is trivial. Or it may be more sophisticated, providing configuration and dependency information to let the submit script determine whether or not the receiving codeline has changes which might conflict with the propagated change, or which should at least require testing and verification before proceeding.

Resulting Context

- Proper ordering of propagated changes can be easily tracked and enforced.
- Some propagation tasks may be delayed waiting for a predecessor task to be propagated. This is deemed acceptable in order to prevent changes from being propagated in the wrong order and direction.
- If *PYOC* is being used, propagation queues guards against developers accidentally propagating other people's changes while propagating their own.
- If the *Codeline Owner* is performing all integrations for the receiving codeline, then

change propagation queues can be very helpful to the codeline-owner for tracking dependencies between submitted changes, even when *Multi-Merging* is used to integrate several change-tasks at once.

- If you have to roll-your-own solution, rather than making straightforward customizations to a tracking system or SCM system, then the implementation and testing effort involved can be quite significant.
- If you implement the propagation queues yourself using regular files (rather than a database), you will need to go to the trouble of using file locking to prevent concurrent updates to the propagation queues.
- You may need to be *very* familiar with how to write scripts using your VC and tracking tools. Unless your need for propagation queues is quintessential, the effort involved may far outweigh the benefit of implementing them from scratch.

Related Patterns *Merge Early and Often, Propagate Early and Often, MYOC, PYOC*

S7. Third Party Codeline

Pattern *Third Party Codeline*

Context You are responsible for maintaining and porting source code from a third party external to your group or organization. It may be that the code is freeware (e.g. emacs, perl, gcc) or simply that you repackage the code (perhaps with some value-added features) from another vendor to customers that use your product/environment. Some of the changes you make may get cycled back to the vendor but some of them won't be. Thus, you frequently have to cope with keeping your own custom versions and/or configuration of the code "in sync" with updated releases from the vendor. You will also likely need to reproduce earlier versions of the third-party code.

Problem What is the most effective integration/synchronization strategy to accommodate vendor updates with your own custom changes while creating as few headaches as possible?

Forces

- You want to minimize the manual effort required to incorporate the latest vendor release with any custom changes you made.
- You need to be able to reproduce present and previous releases of your customized version.
- You need to be able to isolate your changes to the vendors code for each release (so can send patches to the vendor if necessary, and so you can retrace what needed to be changed)

Solution Use your version control system (VCS) to archive both the versions of the software you receive from the vendor, as well as the versions you deliver to your customer. Use the branching facility of the VCS to track separate but parallel branches of development for the vendor’s code, and your customized versions of the vendor’s code. When the vendor code is received, make it the next version in the vendor branch and then merge the code from that branch into your customized branch.

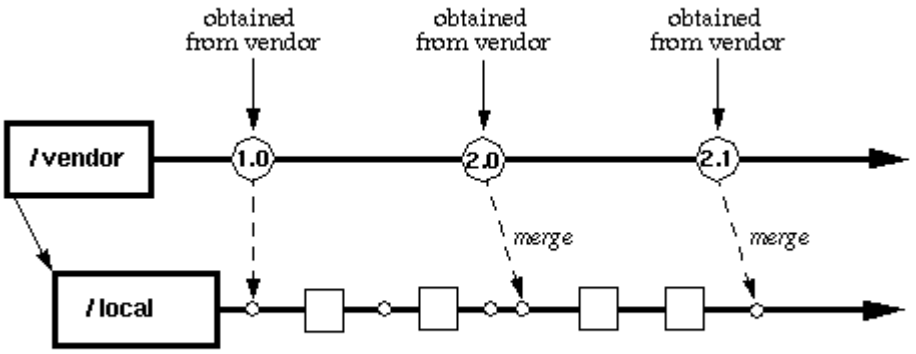


Figure S7a: A separate Vendor Line for pristine versions of 3rd Party Code

Resulting Context

- You can easily reproduce prior versions of your own releases as well as those of the vendor.
- Customization differences can easily by isolated and reproduced so you can see what you had to change for a given release.
- Differences between vendor releases can easily be isolated and reproduced to see what the vendor changed from release to release
- By tracking customization changes on a separate "branch" from vendor changes, you are basically applying a "divide & conquer" approach of orthogonalization: instead of one big change, you logically partition it into vendor changes and custom changes from a common base version. This reduces merge complexity.
- The resulting project "version tree" reflects the real-world development path relationships between the vendor and your group.
- Requires more storage space than simply keeping one source tree or one set (branch) of versions of the source tree.
- Requires the oft-despised merging of parallel changes. There are many people who feel that "merging is evil!" However, in this case, you are not the one who controls the development of the code. You are at the mercy of the 3rd party supplier for this. The best you can hope for is that they incorporate *all* of your changes into their code-base. Thus, merging is really unavoidable here.

Related Patterns This pattern is really a variant of *Parallel Maintenance/Development Lines*. Here, the maintenance line and the development line are distributed across two different sites and one of the needs to be "replicated" or "mirrored" at your site.

S8. Inside/Outside Lines

Pattern ***Inside/Outside Lines***

Aliases *Internal/External Lines, Local/Remote Lines, Central/Remote Lines*

Context Your project must allow outsiders to modify source code for maintenance and/or development. However, those outsiders aren't all at the same location, or even at a handful of select locations. They may be from all over the globe and any one of a number of sites. External developers will have centralized access to the codebase via some kind of remote connection. This is common for open-source software development projects with a large

but core group of primary developers (e.g. Perl, Apache, Linux, etc.)

Problem How do you give developers the access they need without permitting a total stranger to (accidentally or maliciously) completely destroy the consistency and integrity of the codeline?

Forces

- The codebase must be made accessible to a respectable number of remotely dispersed people, not all of whom you can afford to implicitly trust.
- The consistency and integrity of the codeline (as well as the codebase) must be maintained.
- Waiting for checkout-locks to be released may be entirely intractable for respectable numbers of geographically dispersed individuals collaborating together.
- Some degree of control *must* be implemented to safeguard against potentially destructive operations and individuals.

Solution Use a separate codeline for "internal" use that is restricted only to trusted individuals at the master development site (the once which controls and administers the repository). Everyone else uses an "external" codeline that may restrict certain operations, or simply serve as a firewall to the internal line. Changes on the external-line are periodically merged to the internal-line by the latter codeline's owner (and perhaps by some of the other trusted individuals). All official, stable baselevels are kept on the internal codeline.

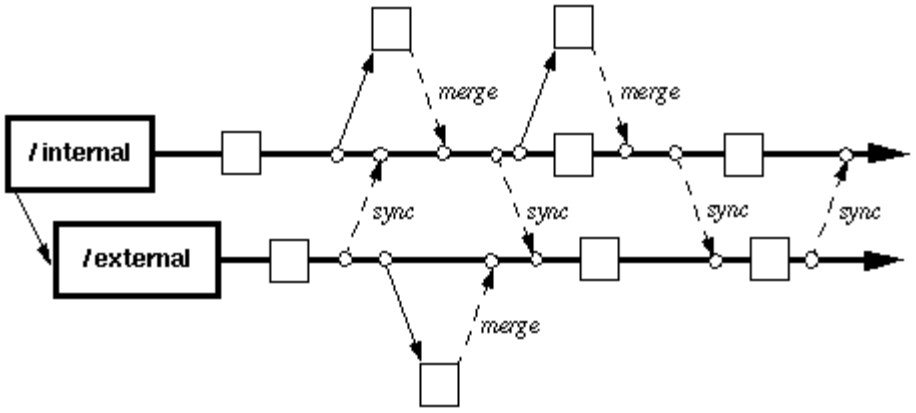


Figure S8a: Separate Inside/OutsideLines provide an extra level of safety from external access to a central repository

Choosing an appropriate codeline-owner for the inside-line should be no more difficult than usual. The same is not true for the outside line since most outsiders don't get to communicate face-to-face very often. Since the external-line serves as the outside gateway for the internal-line, select one of the trusted individuals for the internal-line to serve as the owner of the external-line. This will help keep the policies for the two codelines in alignment with each other.

Resulting Context

- Developer's on the outside-line have the necessary access to the repository without having to wait on one another (possibly across disparate time zones) for checkout-locks to be released.
- The outside line serves as a "firewall" to protect the inside-line from unauthorized or unintended changes. Although the outside-line itself may fall into a state of disrepair, the inside line is insulated from such disasters.
- Extra merging is required to propagate changes to the inside-line and verify their

correctness, but this is deemed essential to preserve the integrity of the primary-line and the safety of the codebase.

Related Patterns

The use of strong *Codeline Ownership* and *Codeline Policy* is especially important for such a geographically dispersed group of collaborating developers. The outside-line will need to be some from of *Relaxed-Access Line* while the inside line will obviously need to employ some form of locking for a *Restricted-Access Line*.

For an added degree of safety/stability (at the expense of more integration overhead), add a *Stable Receiving Line* which is reserved for the sole purpose of receiving propagations of stable baselines from the internal line.

This pattern is similar to *Remote Line* but the external line isn't for a particular remote-site; it's for *all* of them! A remote-line is also better suited for replicated (or "mirrored") repositories instead of centralized access to a lone repository.

Acknowledgements

The authors would like to give special thanks to the following people for their significant contributions to *Streamed Lines*:

- Chris Seiwald and Laura Wingerd of Perforce Software, for sharing their drafts of, and considerable expertise with, high-level SCM best practices
- Doug Lea, our shepherd for PLoP'98, and concurrent programming "guru" extraordinaire
- Steve Vance, for sharing his drafts of advanced SCM branching strategies
- Linda Rising, David Delano, Neil Harrison, and all the other folks responsible for putting together the ChiliPLoP'98 conference, which gave the four of us the opportunity to meet face-to-face and collaborate in the same room at the same time.

References

[Appleton97] Brad Appleton; *Patterns and Software: Essential Concepts and Terminology*; Object Magazine Online http://www.sigs.com/omo/, May 1997, Vol. 3 No. 5; http://www.enteract.com/~bradapp/docs/patterns-intro.html

[Beedle97] Michael A. Beedle; "*cOOherentBPR - A pattern language to build agile organizations*"; in **PLoP/Allerton Park 1997 Proceedings**; Washington University Technical Report #wucs-97-34

[OrgPats] *Organizational Patterns Wiki Web*; http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns

[Berczuk95] Stephen P. Berczuk; *Patterns for Separating Assembly and Processing*; in **Pattern Languages of Program Design**, James O. Coplien, Douglas C. Schmidt (Ed.), Addison-Wesley, 1995, pp. 521-528

[Berczuk96] Stephen P. Berczuk; *Organizational Multiplexing: Patterns for Processing Satellite Telemetry with Distributed Teams*; in **Pattern Languages of Program Design 2** J. Vlissides, J. Coplien and N. Kerth, editors; Addison-Wesley, 1996, pp. 193-206

[POSA] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerald, Michael Stal; **Pattern-Oriented Software Architecture: A System Of Patterns**; John Wiley & Sons, 1996

[Berczuk97] Stephen P. Berczuk; *Teamwork and Configuration Management*; C++ Report, Vol. 9 No. 7, July/August 1997

[Perry98] Dewayne E. Perry, Harvey P. Siy, Lawrence G. Votta; *Parallel Changes in Large Scale Software Development: an Observational Case Study*; in **Proceedings of the 20th International Conference on Software Engineering** (ICSE 20); ACM Press, Kyoto Japan, April 1998

[Atria95] *The Problems of Parallel: Overcoming the Obstacles in Team-Based Software Development*; Whitepaper by Atria Software, 1995

[Tichy85] Walter F. Tichy; *RCS - A System for Version Control*; in *Software Practice and Experience*; Vol. 15 No. 7, July, 1985

[Rochkind75] Marc J. Rochkind; *The Source Code Control System*; in *IEEE Transactions on Software Engineering, Vol. SE-1 No. 4, December, 1975*

[Leblang94] David B. Leblang; *The CM Challenge: Configuration Management that Works*; in

Configuration Management (Trends in Software series, vol. 2); edited by Walter F. Tichy; John Wiley & Sons, July 1994, pp. 1-37

[Seiwald96] Chris Seiwald; *Inter-File Branching: A Practical Method for Representing Variants*; Sixth International Workshop on Software Configuration Management (I-SCM6), Berlin, Germany, March 1996; in **Software Configuration Management: Selected Papers of the ICSE SCM-6 Workshop**; edited by Ian Somerville; Springer-Verlag, 1996, pp. 67-75

[CVS] Per Cederqvist et. al.; **Version Management with CVS**; Cyclic Software

[GoF] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, **Design Patterns: Elements of Reusable Object-Oriented Software**, Addison-Wesley, 1995

[Cope95] James O. Coplien; "*A Generative Development Process Pattern Language*"; in **Pattern Languages of Program Design**; James O. Coplien, Douglas C. Schmidt (Ed.); Addison-Wesley, 1995, pp. 178-237

[Cockburn97] Alistair Cockburn; **Surveying Object-Oriented Projects: A Manager's Guide**; Addison-Wesley, 1997

[Schmidt96] Douglas C. Schmidt and Steve Vinoski; *Comparing Alternative Programming Techniques for Multi-Threaded Servers*; C++ Report, Vol. 8 No. 2, February 1996

[McKenney95] Paul McKenney; *Selecting Locking Designs for Parallel Programs*; in **Pattern Languages of Program Design 2** J. Vlissides, J. Coplien and N. Kerth, editors; Addison-Wesley, 1996, pp. 501-531

[Lea96] Doug Lea; **Concurrent Programming in Java: Design Principles and Patterns**; Addison-Wesley, October 1996

[Kruchten95] Phillipe Kruchten; *The 4+1 View Model of Architecture*; in *IEEE Software*, Vol. 12 No. 6, November 1995, pp. 42-50

[Davis97] Margaret J. Davis and Roger B. Williams; *Software Architecture Characterization*; in **Proceedings of the 1997 Symposium on Software Reusability** (SSR'97); Medhi Harandi, editor. ACM Press, May 1997, pp. 30-38

[Kriha97] Walter Kriha, Daniel Kesch, Stephan Pluess; *Architectural Structures for Large Systems Design*; Position paper for Workshop *Exploring Large Systems Issues*; OOPSLA'97, Atlanta, USA, October, 1997

[Feiler91] Peter H. Feiler; *Configuration Management Models in Commercial Environments*; **SEI Technical Report CMU/SEI-91-TR-7**, March 1991

[Wingerd98] Laura Wingerd, Chris Seiwald; *High-level Best Practices in Software Configuration Management*; Submitted to the Eighth International Workshop on Software Configuration Management (I-SCM8), Brussels, July 1998; (also presented at the 1998 Perforce User's Conference, June 1998)

[Vance98] Stephen Vance; *Advanced SCM Branching Strategies*; 1998 Perforce Users Conference, June 1998, Oakland, CA (P4UC'98)

[Fierro98] Doug Fierro, Rational Software; *ClearCase Branching Strategies*; 1998 Rational User's Conference, June 1998, Orlando, FL (RUC'98); Change and Configuration Management Track

[McKeen97] Andy McKeen, Rationale Software; *Managing Software Projects in a Parallel*

Development Environment; 1997 ClearCase International User's Group conference (CCIUG'97)

[White95] Brian White, Atria Software; *Is `/main/LATEST` Too Dynamic? Software Integration Strategies Using ClearCase*; 1995 ClearCase International User's Group conference (CCIUG'95)

[Atria94] Atria Software; *How Atria Uses ClearCase* 1994 ClearCase International User's Group conference (CCIUG'94); (also presented at CCIUG'95, CCIUG'95, and CCIUG'97)