

CODELINE MANAGEMENT FOR EVOLUTIONARY DEVELOPMENT

Anders JOHNSON
WIS Technologies
ajohnson@wischip.com

Abstract: *Carrying out evolutionary development presents certain source configuration management challenges. A methodology for addressing these challenges using Perforce is presented. It is found that the presented methodology can be applied to concurrent project development as well. The methodology is supported by a set of freely available Perl scripts.*

1 Evolutionary Development

The advantages of evolutionary development in its various forms are widely recognized.^{1 2 3 4 5} Specifically, substantial simultaneous improvements over traditional methodologies in quality, productivity and project risk have been observed.⁶ In short, evolutionary development dictates that a functioning system skeleton is integrated as early as possible, and features are then added piecemeal while maintaining near-production quality throughout.

Frequent, thorough quality assessments at various levels of integration are essential for evolutionary development. Although automating such tests requires a substantial up-front commitment, repeated manual testing is extremely uneconomical and unreliable. Failure to automate invariably results in a dramatic reduction in the frequency and thoroughness of quality assessment, thereby undermining evolutionary development.

2 Codeline Integrity

Typically, the probability that a given untested change results in a catastrophic failure (*e.g.* failure to compile or inability to perform basic functions) exceeds 10%, and it is sometimes much greater. When a change received from another contributor results in a catastrophic failure, it then becomes difficult or impossible to continue development, because the recipient can no longer test his own changes. Furthermore, it may be difficult to isolate the change and/or file containing the catastrophic defect. It is therefore advisable for a change to remain invisible to the general community of contributors until it is at least “smoke” tested.⁷

With Perforce, there are two basic approaches used to avoid receiving catastrophic defects. Under the first approach, which I call the *gated update discipline*, contributors are instructed to **p4 sync** only to a particular label designated to represent the latest known-good change number. Under the second approach, which I call the *gated submission discipline*, contributors are instructed to **p4 submit** only known-good changes. In either case, “known-good” is defined as passing some automated regression test. Either approach can benefit from submit trigger and/or daemon automation, but a sufficiently disciplined organization could use ordinary scripts to accomplish the same thing.

The gated update discipline has the advantage that untested changes can be sync'ed in by an adventurous contributor on a per-file basis. Also, it permits a subsequent workspace to be submitted while the regression is running, provided that each submitted file is up-to-date. However, it provides no means of discriminating among different untested changes to the same file. Furthermore, there are no inherent safeguards to prevent the codeline from failing the regression for an unbounded period of time, while contributors gradually abandon the “known-good” label as it becomes frozen in the distant past.⁸ Also, there is no guarantee that a combination of working

¹ Brooksby, Richard, “[Product Quality through Change Management](#),” 1999 Perforce User Conference.

² Brooks, Frederick P., “[The Mythical Man-Month](#),” 2nd ed., pp. 200-201.

³ Lippert, Martin, *et al.*, “[eXtreme Programming in Action](#),” §§2.4, 2.5, 2.10.

⁴ Lakos, John, “[Large-Scale C++ Software Design](#),” §4.8.

⁵ Yourdon, Edward, “[Death March](#),” §5.6.

⁶ McConnell, Steve, “[Rapid Development](#),” §§4.3, 7.5, 7.8, 18, 20, 21.

⁷ *Smoke test* is defined in *Ibid.*, §18.

⁸ I have seen it happen more than twice, although we were using CVS rather than Perforce at the time. It takes only a few bad apples to spiral the development away from a working state. They'll complain that the “Methodology

submissions is itself a working configuration. This approach is therefore not particularly well-suited for evolutionary development.

The gated submission discipline has none of the identified disadvantages of the gated update discipline. However, it requires that all files relevant to the regression (including files *not* being submitted) be up-to-date when one submits, and therefore the frequency of successful submission is strictly limited by the regression latency. Furthermore, it relies on branching for selectively sharing untested changes. Branching presents its own unique set of challenges, not the least of which is educating contributors regarding its proper usage. Nonetheless, it has been found that branching is simply too useful to ignore,⁹ and thus a viable methodology must account for it sooner or later. Another subtle advantage of this second approach is that the earliest contributor to submit a *working* change (as opposed to a change of any level of quality) is burdened with the least amount of merging effort, and thus there is an inherent incentive for every contributor to attain incremental progress as rapidly as possible.

For these reasons, we have decided to focus on the gated submission discipline, although it is acknowledged that the selection between the two approaches remains contentious.

3 Multi-Contributor Modifications

Having settled on the gated submission discipline, the need for branches quickly becomes apparent. Suppose that in order to pass the regression, a given proposed modification (or *job* in Perforce parlance) requires submissions from multiple contributors. For example, the job may span multiple disciplines (such as hardware and software), or scheduling constraints may require its development to be parallelized among developers. In this case, no single contributor can submit his changes to the mainline, because they are all known to fail the regression. In theory, this problem could be addressed by sharing a workspace, but sharing workspaces is well-known to be ill-advised.^{10,11}

The recommended technique for resolving this dilemma is to create a new codeline to accumulate the contributions to the job. When the job is complete, it is integrated and submitted into the mainline. Such a codeline is called a *development codeline*.¹²

Of course, not every codeline will be a development codeline.¹³ It is therefore important that the Perforce usage conventions that an organization adopts be sufficiently lenient to accommodate other codeline roles as well. The conventions presented herein satisfy this requirement.

4 Per-Codeline Regression

Different codelines need to have different regressions. In the case of a development codeline, the regression might consist of basic lint checks, or it might not even exist at all.

In order to ensure that the correct regression for the codeline to which the contributor submits is applied, we elected to formalize the concept of a codeline well beyond what is dictated by Perforce itself. While this formalism necessarily diminishes the power of Perforce somewhat, we concluded that something had to be done to rein in the chaos.

First, we decided that each workspace must correspond to exactly one codeline. The codeline to which the workspace corresponds

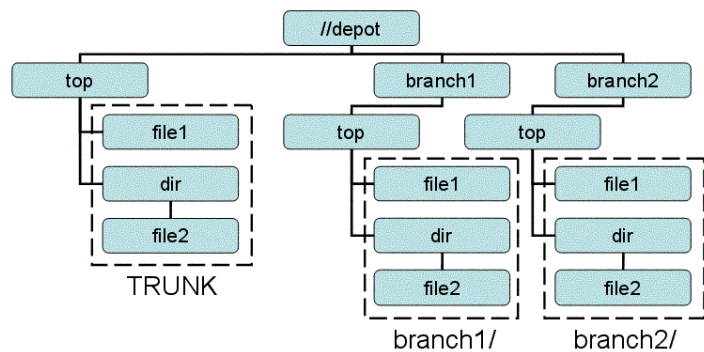


Figure 1. Depot Structure

Police” is trying to get in their way, but ultimately it’s not evolutionary development any more when the system hasn’t been successfully compiled in a month. We abandoned this approach shortly before switching to Perforce.

⁹ Vance, Stephen, “[Advanced SCM Branching Strategies](#),” 1998 Perforce User Conference.

¹⁰ [Ibid.](#)

¹¹ Wingerd, Laura, *et al.*, “[High-level Best Practices in Software Configuration Management](#),” 1998 Perforce User Conference.

¹² The terms *branch* and *codeline* are more or less interchangeable. I prefer to reserve *codeline* for whole-workspace branches, whereas *branch* can refer to anything that results from **p4 integrate**.

¹³ For a taxonomy of codeline roles, see Vance, [op. cit.](#)

is embedded as an *extension field* within the Perforce client record's `Description` field.

Second, we decided that each codeline must have exactly one parent codeline (except for the mainline, or “TRUNK,” which has no parent codeline). The parent codeline is an extension field in the Perforce branch record.

Third, in order to avoid name collisions between codelines and top-level directories (which would otherwise be commonplace), we adopted the convention that the depot be structured as a hierarchy of codelines whose leaves are directories named “`top`,” each of which contains the directory hierarchy of the corresponding codeline. (See Figure 1, above.) This convention precludes any codeline name from containing “`/top/`.”

Finally, we decided that each codeline's regression shall be described by a designated file within the codeline itself, such that the regression could be modified using normal Perforce mechanisms. For better or for worse, any contributor can modify the regression, so it is advisable to appoint a “regression czar” to monitor any such changes, for example, using a review daemon.

5 Per-Codeline View

The need for each codeline to have its own view is perhaps less obvious. Because different codelines have different scopes, it is advisable to minimize the configuration management overhead of each codeline by limiting its contents to files that are relevant to the work being done on the codeline (minimally, to the files needed for running its regression).

We decided, more or less arbitrarily, to accomplish this goal by restricting the client view rather than the branch view. This approach has the minor advantage of allowing relevant but rarely-needed files to be stored in the codeline without causing the typical workspace to view them.

We decided to use a designated file within the codeline to describe the default view for each codeline, as it dramatically reduced the amount of duplicated effort in setting up clients.

We further decided that a file's location within the codeline would correspond exactly to its location in every workspace, as this convention improved reproducibility both among various workspaces and among various codelines.

It is to be noted that the preceding conventions ensure that a source configuration can be *definitively* specified with nothing more than a codeline name and a change number.

Additionally, a script was developed to generate a symbolic link called “`.p4top`” in every workspace directory that points to the workspace root. Running the script allowed files in a given workspace to portably refer to each other using *pseudo-absolute* paths beginning with “`.p4top/`.” Because every “`.p4top`” in the workspace points to the same directory, a pseudo-absolute path can usually be used where an absolute path would normally be required, such as the value of a **PREFIX** configuration variable in an install script.

6 Sharing Infrastructure among Codelines

The regressions and views of one codeline will often benefit from sharing both infrastructure (*e.g.* scripts) and content (*e.g.* test and file lists) with other codelines. For example, it is often advisable for a parent codeline to derive its view from the union of the views of its children, and to derive its regression from the union of the regressions of its children.

In order to facilitate such sharing, the workspace location of the *configuration directory* (which contains the regression and view information) is codeline dependent, such that the configurations of all the other codelines can be in view simultaneously. In particular, we adopted the convention that the configuration directory is given by “`workspace-root/CFG/codeline-name/top`.”

Sharing of configuration information among codelines can be accomplished through a variety of techniques, such as symbolic links and generation scripts. See §11, “[Concurrent Projects](#),” for a more complete list of ideas.

7 Staying in Sync

In many cases (including the case of a development codeline), we want to maintain the property that if Perforce thinks that there is nothing to merge between a child and parent codeline, then the contents of those codelines are identical. This property is surprisingly difficult to uphold, due to integration edge effects that occur too frequently in practice to discount.¹⁴

¹⁴ Johnson, Anders, “P4 Integrate Flaws,” <http://www.andersjohnson.com/perforce/p4integ>.

Except where the child codeline explicitly waives this requirement in an extension field (in particular, for release codelines¹⁵ and for vendor codelines¹⁶), we support it using the following discipline:

- Codeline integration occurs only from parent codeline to child codeline, or vice-versa.¹⁷
- Reverse integration (from child to parent) may occur only after the child codeline contains all of the parent codeline's changes. Rejecting a change from the parent codeline (*e.g.* **p4 resolve -ay**) is tantamount to proposing that the change be reverted.
- Reverse integration is performed with **-dft** and resolved with **-at**, thereby effectively copying the child codeline into the parent codeline.

Regrettably, this discipline precludes some useful practices, such as accumulation branches,¹⁸ and thus cannot be followed religiously. However, any resulting problems can be addressed so long as special care is taken whenever this discipline is violated. In practice, this typically entails manually rectifying the integration history (using **p4 integrate** with **p4 resolve -ay**) after indirect integration (using **p4 integrate -i**, often with a specified revision range).

8 Switching Codelines

Sometimes, through carelessness or poor coordination, a contributor finds himself with valid changes stranded in a workspace where they cannot be submitted without failing the applicable regression. Discarding the changes would be costly. Copying the affected files to a workspace on another codeline is likely to cause difficulties with regression and/or with future merging.

To deal with this problem, a script was developed to generate a patch file based on the output of **p4 diff** and **p4 opened** in one client, and to apply that patch file to another workspace.

9 Hierarchical Regression

One of the most important advantages of this methodology in promoting evolutionary development is that it can be used to break the “sound barrier” of one submission per regression, because every codeline can be running its own regression at any given time. While this advantage provides little benefit if the regression is bandwidth limited (for example, if job queuing software is used to fully utilize every machine), it can be a productivity boon when the regression is latency limited.

For example, the mainline might have a regression that runs for 4 hours. However, the development can be broken up into feature codelines that each have a regression consisting of a 2-minute compile and smoke test followed by a 1-minute feature test. Development codelines are derived from each feature codeline as needed. (See Figure 2, right.) Each feature codeline can accept a submission every 3 minutes. Every few days (typically on a staggered schedule), each feature codeline forward integrates from the mainline, passes its own 3-minute regression, and then reverse integrates into the mainline, undergoing the 4-hour mega-regression.

Under this scenario, submission latency is minimal, submission bandwidth is high, and each feature codeline receives a thorough regression every few days, so it never has time to diverge very far from near-release quality. Furthermore, feature codelines are periodically synchronized with each other (see §7, “[Staying in Sync](#)”), but they never infect each other with un-regressed changes.

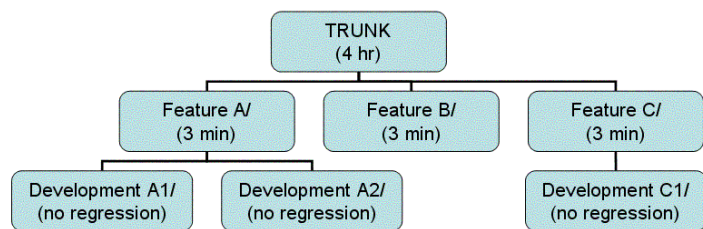


Figure 2. Hierarchical Regression

¹⁵ Vance, *op. cit.*

¹⁶ Perforce Technical Note #15, “[Creating a vendor branch for third-party source code.](#)”

¹⁷ Laura Wingerd calls this “tributary flow.”

¹⁸ Vance, *op. cit.*

10 Reference Builds

We decided that pre-submission regressions would always occur within the workspace being submitted. The first reason for this decision is that it would be awkward (but not impossible; see §8, “[Switching Codelines](#)”) to reproduce the submitted source configuration elsewhere before it exists in the depot. The second reason is that one can usually rely on the build system to compile the system correctly with minimal latency based on pre-existing generated files. Ideally, one could even have the build system figure out which tests need to be re-run.¹⁹

However, there are drawbacks to this approach. For example, the build system might rely on a source file that exists in the workspace being submitted, but that hasn't been added to the Perforce depot. Or maybe the regression passes only because of some unique environment setting. Or maybe the build system is broken, and it neglected to recompile a recently broken source file.

To detect such problems, it is advisable to periodically check out a clean source tree as a pseudo-user with a minimal environment, and then run the regression. I call the resulting workspace a *reference build*.

A reference build can be deleted after it succeeds, but leaving the most recent reference build on-line has the advantage that future workspaces can be cloned from it, in conjunction with **p4 flush**. Such a cloned workspace can then be sync'ed to the latest source versions, and then the build system can update the generated files only as needed. This technique is especially beneficial for a project with a long clean-build latency.

Because workspaces can be created without clean building, it is practical to employ practices that enhance correctness at the expense of clean-build latency. For example, evolving tools (such as GCC and Perl) can be stored (as source code) in the depot and built in workspaces, such that older configurations will continue to work, even if they rely on deprecated tool behavior.

11 Concurrent Projects

Some organizations simultaneously develop multiple products with distinct feature sets, but having a substantial amount of core functionality in common. Such a strategy is particularly common in hardware development organizations, because the unit cost is usually important and feature-set dependent.

In my experience, the most successful attempts at multiple related concurrent projects have used a single maximally-shared code base.²⁰ This approach is known to work well whether or not branching is used.

One distinct advantage of this approach is that a more mature project (perhaps even a project that has already been completed or cancelled) can serve as a test platform for changes submitted on behalf of an incipient project.

Typically, a variety of techniques for managing variations among related projects are available. For example:

- Use a search path to select overridden files from an overlay directory.
- `#include` a header file from a shared location.
- Read a source file, object file or library from a shared location.
- Symbolic links to shared files in a duplicated directory.
- `#ifdef` in a shared source file to express variations.
- Leave code as-is, even if not all projects require it.
- Design patterns.²¹

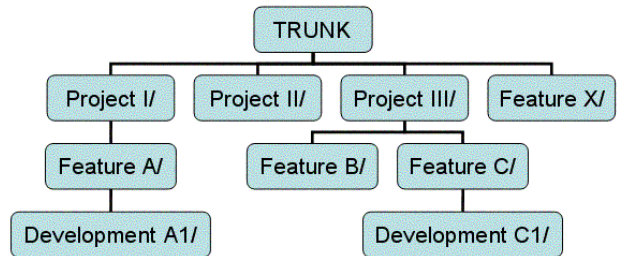


Figure 3. Concurrent Projects

¹⁹ I've never had the courage to trust my build system *that* much, but maybe you do.

²⁰ The party line seems to be that Perforce inter-file branching takes care of it for you, but I remain skeptical. The fact that it requires somebody other than the change author to determine which changes are intended to be propagated to another project seems fraught with danger. Furthermore, if there are more than 2 projects, then the potential for replicated merging effort is daunting. On the other hand, if it is guaranteed that there will never be a need to merge among branches, then by all means branch away.

²¹ Gamma, Eric, *et al.*, “[Design Patterns](#).”

- Decompose an entity into invariant parts and variant parts.
- Filter a common source file with a Perl script to produce several variations.
- Make the project requirements more similar.
- Use inter-file branching²² to generate a new variation in a different workspace location (as a last resort).

Normally, some combination of such techniques must be used to express the necessary variations with minimal maintenance effort. On one successful project, each and every one of the listed techniques was used at least once.

Making a separate unrelated copy of a related file is to be strictly avoided, because it provides no support for propagating beneficial changes. Novices tend to grossly underestimate the impact of this drawback on development effort and/or product quality.²³

By using a shared code base, the configuration management problem reduces to one of managing a single *super-project* that is somewhat larger than the largest project. It is then possible to apply the “[Hierarchical Regression](#)” approach (see §9) to break the super-project into shared feature codelines and project codelines, and then further into project feature codelines if necessary. (See Figure 3, above.)

12 Edge Effects

The methodological choices that we made resulted in a few interesting and partially unanticipated consequences that require special attention.

12.1 Mixed Views

A workspace containing files from multiple codelines is explicitly eschewed because such a configuration is difficult to describe, and therefore difficult to reproduce. However, this scenario could be useful for legacy compatibility testing in particular.

The recommended work-around is to use inter-file branching in the source codeline to duplicate the desired versions to another workspace location, preferably one that has the version name in the directory path. One can then use the normal codeline-to-codeline integration mechanisms to convey those files to the target codeline.

If the need for using this technique occurs frequently, then one can avoid the latency associated with codeline-to-codeline integration by combining the source and target codelines. However, because the view and regression of the combined branch will be the union of the views and regressions, respectively, of the source and target branches, and because the combined branch cannot have multiple simultaneous regressions, it may be inadvisable to do so.

12.2 Forward Integration Failures

It is possible for a child codeline's regression to fail after forward integrating from its parent codeline. The principal causes of such a failure are that the parent codeline's regression doesn't include the child codeline's regression, or that some of the parent codeline's changes are incompatible with changes in the child codeline.

If the regression failure stems from a defect, then it is usually best to correct the defect at its source. Determining which codeline is “defective” is sometimes a subjective determination that requires coordination and planning. On the other hand, if the failure stems from inadequate merging, then it is usually best to select the “e” option to **p4 resolve**, and/or to **p4 edit** additional files as necessary. If the merging edits are unmanageably complex, then one might prefer to submit some of them to a grandchild codeline, and reverse integrate them into the child along with the forward integration from the parent, prior to submitting to the child.

12.3 Inter-file Branching

Inter-file branching (that is, integrating between different workspace locations within the same codeline) requires special attention, because any future integration between the same file locations must occur on the same

²² I prefer to reserve the term *inter-file branching* for integration between different workspace locations, as opposed to ordinary codeline-to-codeline integration. “Inter-File Branching” is a trademark of Perforce Software.

²³ When confronted with dire predictions of “#ifdef hell,” I have found it useful to point out that it is straightforward to bifurcate at any time should sharing become unmanageable, whereas once copies start evolving separately, it rapidly becomes prohibitive to re-merge them intelligibly.

codeline, or else the integration history between them is lost. This problem is partially addressed by the new behavior of **p4 integrate -i** in Perforce Release 2002.2, but indirect integration is still not as well-behaved as direct integration.²⁴

If the file locations have already been integrated on a codeline whose regression doesn't pass after a subsequent integration, then it is recommended that either of the following techniques be employed:

- Use one of the solutions described in §12.2, “[Forward Integration Failures](#).”
- Do an indirect inter-file integration in a child codeline.²⁵ After the child codeline is reverse-integrated, **p4 integrate** with **p4 resolve -ay** in the original codeline to rectify the inter-file integration history.

12.4 Reverse Integration with Inter-file Branching

A particularly undesirable situation arises when a file affected by reverse integration is the source of an inter-file branch, the target file location is not in the child codeline’s view, and the parent codeline’s regression requires the inter-file integration to occur before the reverse integration is submitted. The recommended solution is to do an indirect integration from the child codeline to the inter-file target in the parent codeline. After submitting to the parent codeline, the inter-file integration history is then rectified using **p4 integrate** with **p4 resolve -ay**.

13 Methodological Automation

A suite of object-oriented Perl scripts and libraries representing approximately 5 person-months of effort has been developed to formalize and support the presented methodology. The package is called “A4,” and it is freely available at <http://a-4.sourceforge.net>.

The source code size of A4 version 0.01777 is shown in Table 1.

Table 1. A4 Code Size

What	Lines
Ordinary Code	10220
Blank Lines	4130
Comments	579
Documentation	6326
Test Code	600
Total	21855

The seemingly excessive size is driven by an emphasis on reusability, extensibility and robust error handling. The system is actually quite efficient at runtime, due to the aggressive use of autoloading.

14 Conclusions

It is generally advantageous to use some form of evolutionary development in lieu of a traditional late-integration methodology. The gated submission discipline is recommended for carrying out evolutionary development using Perforce. The gated submission discipline requires that codeline branching be a regular part of the development process, and therefore every contributor must understand how it works. A variety of conventions regarding the structure and semantics of each codeline should be followed in order to formalize codeline policies, to avoid mutually incompatible practices, and to maintain quality.

One role of codelines is to enable parallel regression of various proposed changes. Periodic reference builds should be conducted in order to detect latent defects and to avoid clean building in workspaces. It is advisable for concurrent related projects to employ a maximally shared code base with minimal use of inter-file branching between different workspace locations. A freely available program called “A4” can be used to automate the management of codelines.

²⁴ Perforce Technical Note #65, “[How are indirect integrations handled?](#)”

²⁵ Manually selecting the revisions to integrate may or may not be required in order to achieve the desired result from an indirect integration.