Managing Vendor Code Customizations with Stream-based SCM

Why Streams Are Easier than Traditional Branches

David P Thomas
dave@accurev.com

Summary

Customizing or extending third party "vendor" source code is becoming increasingly common especially with the availability of open-source software. Building upon existing code increases your time to market and lets a group of experts elsewhere develop the foundation. Vendors typically provide frequent patches and new features in the form of vendor releases. Managing the incorporation of vendor releases alongside customizations requires an additional layer of configuration management. Traditional branch-based software configuration management (SCM) tools require an unnecessarily complex branch and merge process. This article describes how stream-based SCM provides a more intuitive and efficient parallel development model for managing customizations to vendor code.

The Challenge

Managing customizations to vendor code requires an additional layer of configuration management to integrate subsequent vendor releases. Vendor code must be imported, merged with the previously imported vendor release[1], merged with a selected set of compatible customizations, and finally merged with one or more active codelines. The challenge is to independently track vendor code and orchestrate selective merging and releasing of custom features with vendor upgrades without jeopardizing or disrupting active codelines.

Why Traditional Branch Models Are Difficult

Traditional branch-based SCM models utilize numerous branches to track in parallel both vendor source and custom modifications. In a typical branch model, mainline represents the centralized development codeline[2], a single vendor branch off of mainline isolates and tracks vendor code, feature branches off mainline isolate custom development work, and releases branches off mainline isolate custom releases. A strict coordination of branch-to-branch merges is required to propagate changes between various combinations of branches without violating branch integrity.

[1] **Vendor merge**: Not necessarily the most recent version of the vendor release.
[2] **Central codeline**: mainline is the source of all branches and the destination of all merges. This prevents creating a difficult to manage staircase model of branching where each subsequent release branch is based of the previous.

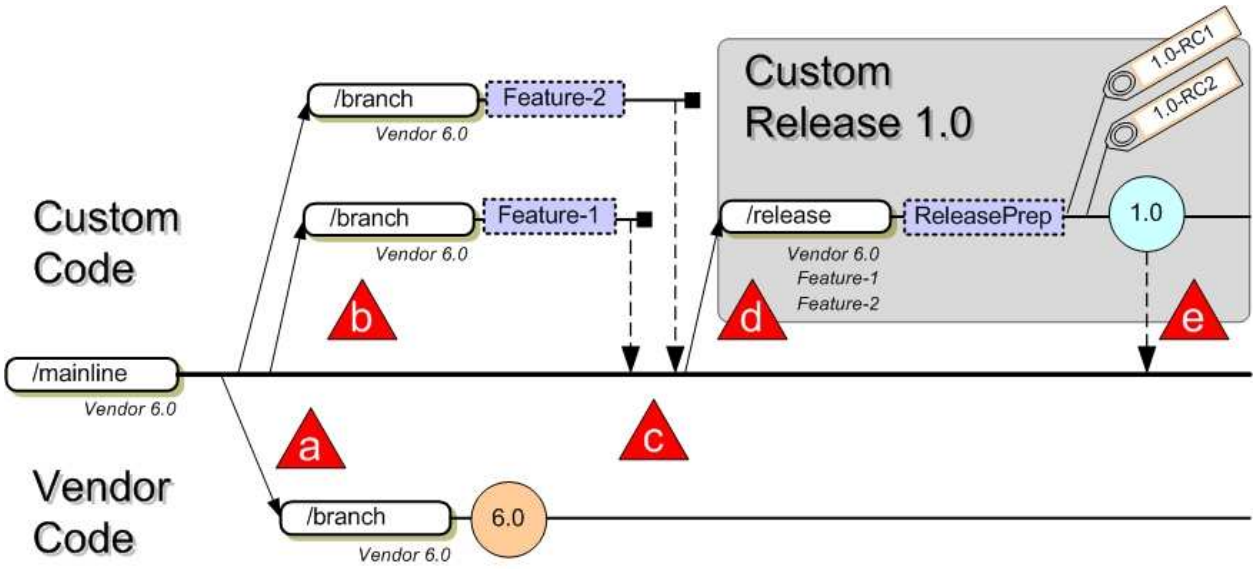

Figure 1 – Traditional branch model for vendor-based custom 1.0 release

The diagram in *figure 1* shows a branch model for a new project tracking a custom release based on vendor code. The project in this example is initialized by importing 6.0 vendor code into the mainline[3]. The vendor code is tracked independently on a vendor branch (label a). Custom features are developed on dedicated feature branches (label b) and eventually merged into the mainline (label c). The custom 1.0 release branch (label d) is used to prepare the release (*e.g.* environment configurations, patches, official testing), provide isolation for release specific customizations, label release candidates, and permit uninterrupted merging of parallel future development onto mainline. Once the custom release branch is tested to satisfaction, it is labeled as "1.0" and merged into the mainline (label e). The custom 1.0 release branch tracks two custom features based on vendor 6.0 code independent of the unmodified vendor code isolated on the vendor branch.

The diagram in *figure 2* is a continuation of the previous branch model highlighting a vendor upgrade, a patch, and a new custom release. Upgrading the vendor code requires importing and merging the 6.1 vendor release into the vendor branch, also known as a "vendor drop" (label f). A vendor-merge branch is created to merge between the upgraded vendor code and mainline containing all current customizations (label g). This branch isolates the mainline from any merge-specific problems such as custom feature incompatibility or file namespace collisions[4]. When the merge is successful, the vendor-merge branch itself is merged into the mainline (label h). A custom release branch is then created to prepare the 2.0 release (label m).

[3] **Vendor import**: Some SCM systems (*e.g.* CVS) have a built-in facility for handling vendor branches. Alternatively, first importing the vendor code to a separate branch and then merging into mainline is sufficient.
[4] **Namespace collision**: The upgraded vendor code may have a new file with the same name as a customized file.

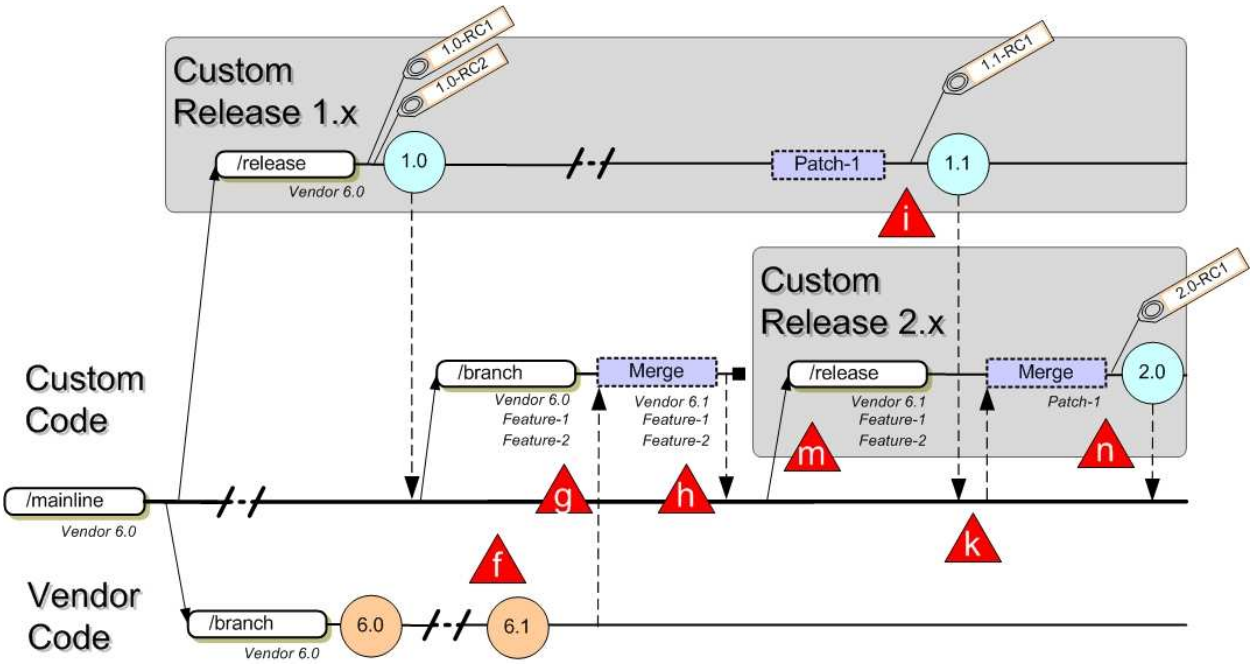

Figure 2 – Traditional branch model for vendor-based custom 2.0 release

In the meantime, a defect is patched on the custom 1.0 release resulting in a custom 1.1 release (label i). This patch is also merged into both mainline and custom 2.0 release branch to prevent regressions (label k). After preparing and labeling the custom 2.0 release, all changes are then merged onto mainline (label n). The custom 2.0 release branch tracks two (previous) custom features and a new patch all integrated with the recently upgraded vendor 6.1 code.

One serious caveat with the above vendor upgrade is that *all* custom features present in the mainline are merged with the vendor 6.1 code in the vendor-merge branch (label g). Integrating only a subset of features with a vendor upgrade requires un-merging custom features! In fact, this *is* what will need to be done in order to preserve mainline as the central development codeline. Taking a step back, *figure 3* shows a highly undesirable branch model to support feature-by-feature vendor upgrade releases. To merge select features with the vendor upgrade, a custom release branch is based off the vendor branch (label p) and merged with selected feature branches (label r). In this scenario, Feature-2 and Patch-1 are merged, but not Feature-1. However, this violates the policy and SCM best practice of mainline being the *central* development codeline and causes a decentralization of release branches! Creating custom release branches off of both mainline *and* the vendor branch quickly turns into an unnecessarily complex web of branching and merging.

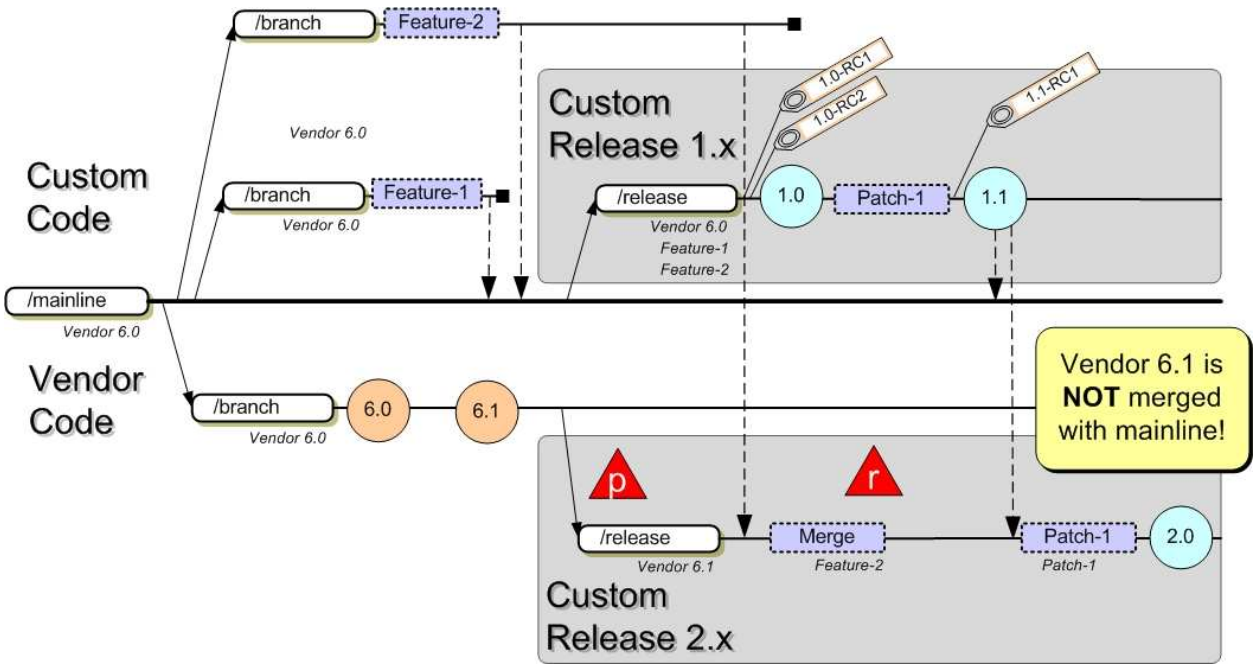

Figure 3 – Traditional branch model for managing feature-by-feature vendor upgrades

Relying on branch-based SCM models to manage customizations to vendor releases requires a complicated orchestration of merges between numerous branches. While this example highlighted only two custom releases, a single patch, and a single vendor upgrade, the situation quickly becomes overwhelming when considering multiple vendor upgrades and multiple custom releases. Lastly, additional complexity arises when considering the propagating of patches between mainline, compatible feature branches, vendor merge branches, custom release branches, or all of the above. It should be clear from this example that the traditional branch based solution quickly becomes unwieldy.

How Streams Make It Easy

Imagine rotating the branch model in *figure 2* clockwise 90 degrees, and allowing automatic inheritance of changes between adjacent branches. Now you are starting to think in streams. A stream based SCM architecture intuitively models parallel development with independent, customizable workflows that make merging simpler with automatic inheritance of changes.

Streams can be thought of as "intelligent" branches individually representing a specific configuration of source code. In more detail, a stream contains a specific version of each and every file visible to the stream. When a developer needs to modify code, they simply create a workspace stream from any stream and instantly have writable access the all the files for that specific configuration. Likewise, when code needs to be deployed, it is simply extracted from a stream to a local directory and packaged.

Streams are organized in a parent-child hierarchy, also called a "stream hierarchy". A stream hierarchy is a tree of streams that intuitively defines a promotion-based software development workflow. Each stream in the workflow represents a stage in the process such as development, integration, QA testing, or SOX auditing. Changes move up the hierarchy by being promoted. To control promotions, streams can be locked by user or role. For example, only members of the release engineer group can promote to or from the QA stream. Streams also have a unique, built-in feature that allows configurations to be automatically inherited down the entire hierarchy from parent to child. This inheritance allows newer file versions higher up in the hierarchy to become automatically available (for update) lower in the hierarchy. Imagine fixing a defect in your 6.1 QA stream and having the patch automatically available to all 50 developer streams *everywhere* in the *sub*-hierarchy.

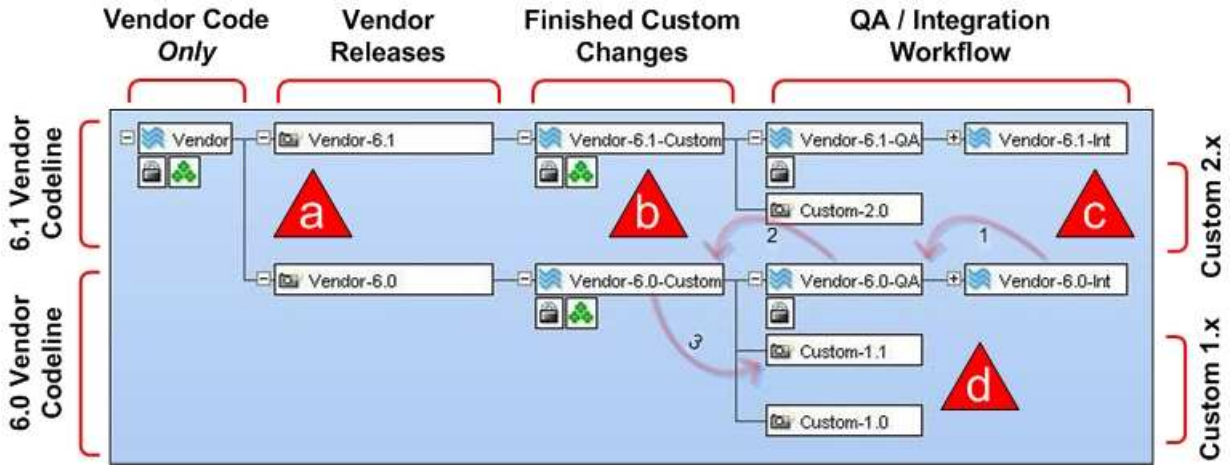

Figure 4 - Stream model overview for vendor-based custom releases

The diagram in *figure 4* shows a stream hierarchy that models the parallel development of custom features based on vendor code. Vendor code is imported and independently managed in the base stream Vendor. Snapshot streams off the base stream are immutable labels that capture the configuration of each vendor release (label a) and serve as named stable bases for version-specific custom development codelines (label b). Each development codeline in this example creates workflow streams named Integration (Int), QA, and Custom to model the development process (label c). New development occurs in workspace streams off the Integration hierarchy (not shown) and is eventually promoted, merged, and tested through both the Integration and QA streams (label c). Finally, the changes are promoted to the Custom stream (*i.e.* production) where a snapshot stream is created to label the official custom release (label d). The green box below a stream indicates that changes are present and inherited downstream. The lock icon signifies that promotions are controlled by user or group. Compared to the branch model, the stream-based model is a more natural organization of vendor releases, custom releases, and custom development workflows.

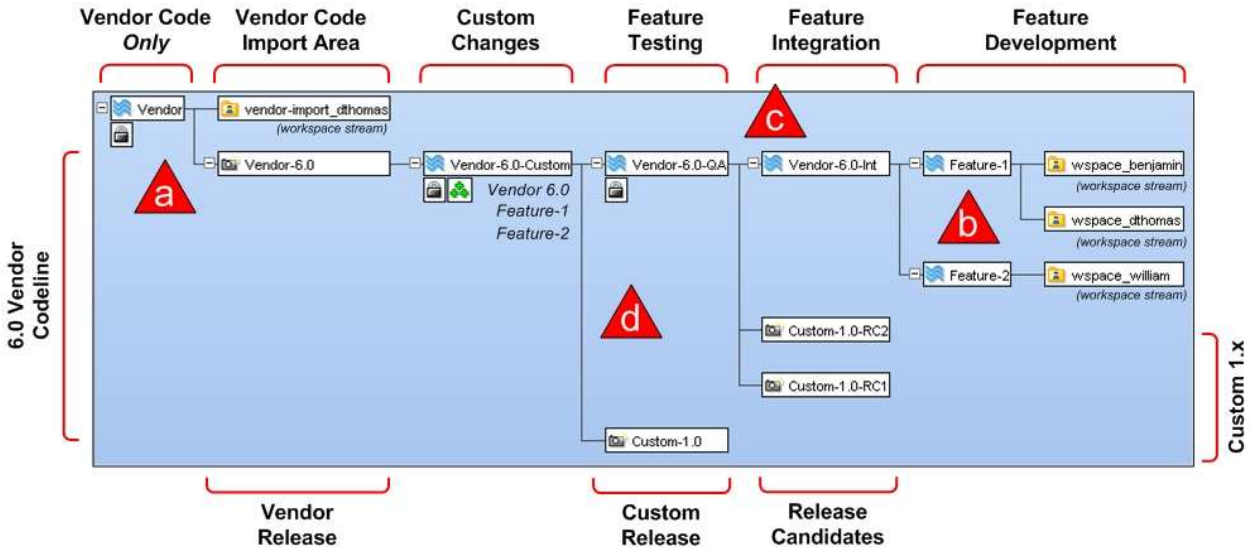

Figure 5 - Stream model for vendor-based custom 1.0 release

Starting a new project based on vendor code requires simply importing the vendor code, creating a vendor release snapshot stream, and setting up a development codeline with streams. The diagram in *figure 5* is a stream structure for a new project tracking vendor code and creating a custom release. The vendor 6.0 source code is imported into the base stream from a workspace stream and a vendor release snapshot stream is created (label a). This snapshot stream serves as a stable basis for the 6.0-based custom development codeline. Custom features are developed in workspace streams and eventually promoted to respective feature streams (label b). Individual feature streams support collaborative development between team members and can also be used for user acceptance testing (UAT). As features become mature, their changes are promoted to Integration to be merged with other features (label c). Keep in mind that features can be promoted when they are either fully or partially complete. The frequency of promotion is directly proportional to the level of continuous integration as promoted changes are immediately inherited by other developers down stream[5]. Inheritance is the key to simplified merging because developers have the option to integrate with other (promoted) features before their own work is completed. Gone are the days of complicated "big-bang" merges at the end of feature development! After smoke testing in Integration, the changes are promoted to QA and subjected to black-box regression testing (label c). Optionally, immutable snapshot streams off of QA can be used to label tested configurations providing full 100% reproducibility of test builds. When testing in QA is complete and the release is scheduled, the changes are promoted to Custom (*i.e.* production) and a snapshot stream is created capturing the custom 1.0 release (label d). So far, the stream hierarchy has organized vendor code independent of a custom release and provided an intuitive workflow for developing, testing, and releasing customizations.

[5] **Inheritance Model**: inheritance is calculated automatically but updating a stream to retrieve the inherited changes is manual. The developer decides when to physically incorporate external changes to their workspace stream.

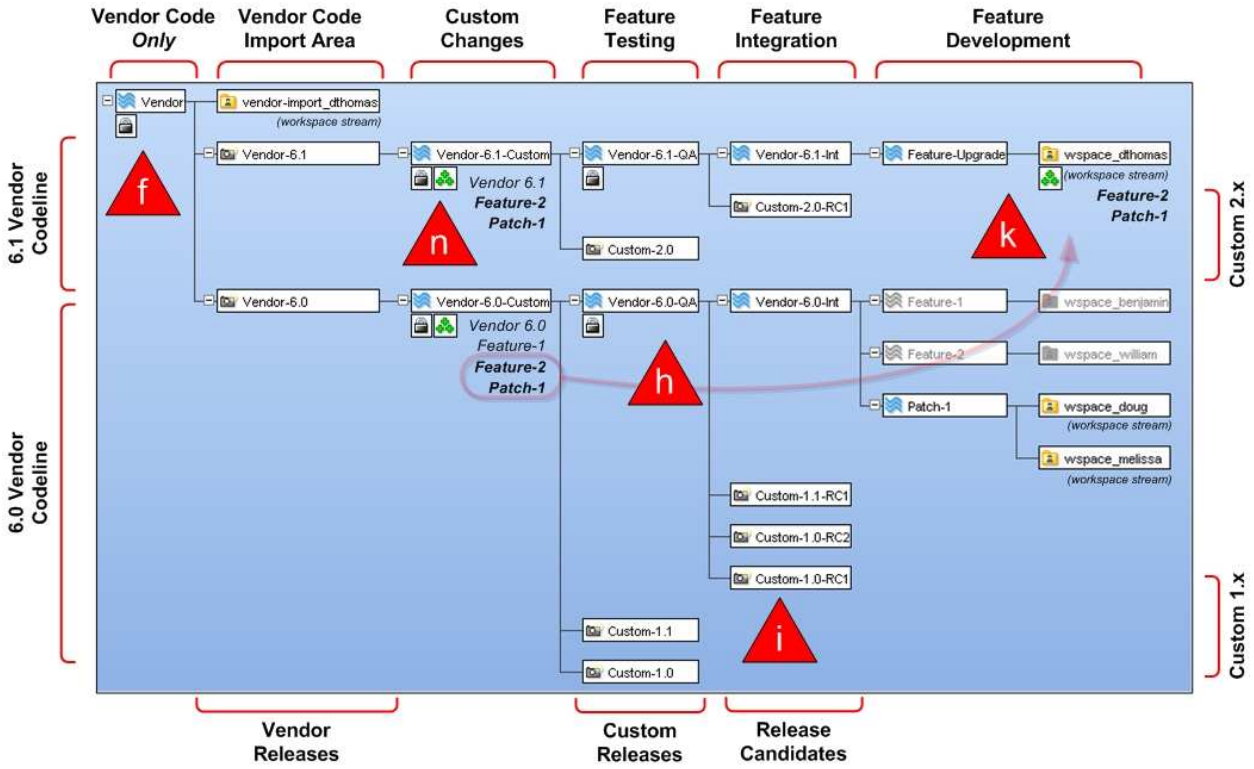

Figure 6 - Stream model for vendor-based custom 2.0 release

The power of this stream model becomes further evident just after the first vendor upgrade. *Figure 6* is a continuation of the previous stream structure highlighting a vendor upgrade, a new custom release, and a feature-by-feature merge. The upgraded vendor 6.1 source code is imported and merged into the base stream from a workspace stream and a 6.1 vendor release snapshot stream is created (label f). This snapshot stream serves as a stable basis for the 6.1-based custom development codeline. Meanwhile, in parallel, a custom patch to the 6.0-based codeline is developed, promoted, and captured in a custom 1.1 release (label i). Now it's time to consider migrating customizations to the 6.1-based development codeline. At this point, all tested and released 1.x customizations are located in the 6.0 "Custom" stream. Which features should be migrated? Stream-based SCM can support promoting changes at either the file level or the feature level using change packages[6]. Promoting by feature makes it very easy to pick and choose features without concerning over which specific files were involved in a given feature development. In this example, Feature-2 and Patch-1 are migrated across codelines by promoting to a workspace stream (label h). Upon promotion, the migrated features will be merged with vendor 6.1 code and tested for compatibility (label k). Performing this merge in a workspace stream prevents conflicts from being inherited and available for update by other developers.

[6] **Change Packages**: using change packages allows developers to contextually group changed files as a single "feature" set. Developers can be prompted to make the association when promoting file from their private stream. In the development workflow, using change packages supports promoting or migrating "by feature" rather than file-by-file.

When the merge is successful, everything is (eventually) promoted to 6.1 Integration, QA, and Custom (label n). A custom 2.0 release snapshot stream is created capturing a feature and patch merge with the upgraded 6.1 vendor code. After removing unused streams, *figure 7* shows the final stream structure modeling two vendor upgrades and three custom releases.

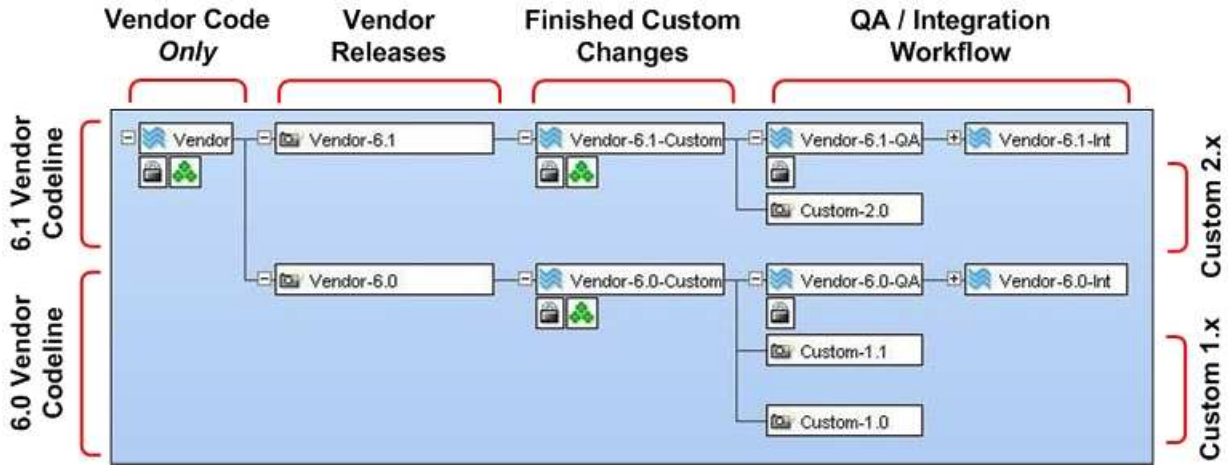

Figure 7 – Final stream model organizing two vendor upgrades and three custom releases

Conclusion

Managing third-party "vendor" based customizations requires an additional layer of configuration management to track vendor upgrades alongside custom releases. To be successful, vendor code must be tracked independent of the dedicated codelines used to develop custom releases. The challenge is to independently track vendor code and integrate vendor upgrades with select customizations while preserving the integrity of the active development codelines. Branch based models utilize numerous branches and require a cumbersome orchestration of merging to be successful. A stream-based model provides a more intuitive solution by using parallel codelines, stream inheritance, feature merging, and a promotion-based workflow. In general, the quality of software progresses from an immature state during development to a mature (tested) state in the release. A stream based model supports defining a methodical workflow that models the natural evolution of software from development to release. Compared to traditional branches, the stream-based model presented in this article provides a more natural way to manage vendor based customizations.

References

Berczuk, Stephen. Software *Configuration Management Patterns*. Addison-Wesley, 2004.

"CVS Vendor Branches". http://ximbiot.com/cvs/manual/cvs-1.11.22/cvs_13.html. July 1 2006.

"Subversion Vendor Branches". http://svnbook.red-bean.com/en/1.1/ch07s05.html. July 1 2006.

Perforce Vendor Branches". http://www.perforce.com/perforce/technotes/note015.html. July 1 2006.