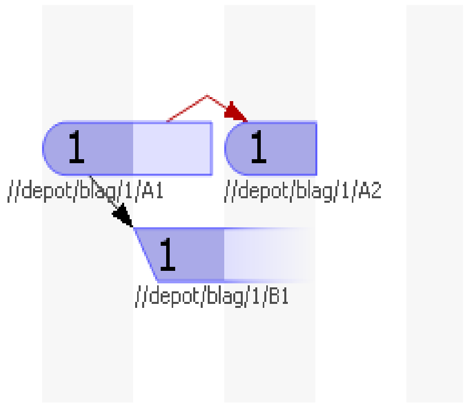
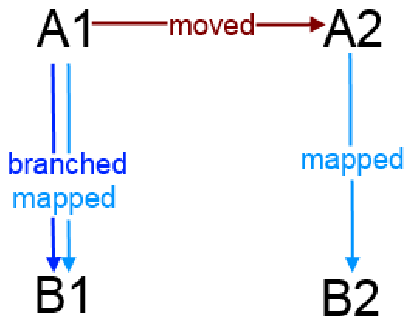
Refactoring and reparenting

In the days before streams, it was commonly accepted that if files had different names in different branches, you would need to set up a branch spec that mapped one set of file names to the other if you wanted to integrate changes between those branches. When we began developing streams functionality, we knew we would need to provide another way to handle refactoring within streams, since the branch view used to merge changes between a stream and its parent is dynamically generated and is supposed to be a relatively simple function of the paths specified for each stream – hence we came up with a system for matching different filename variants within a source and target to each other and setting up resolves between them. When designing this aspect (and others) of the streams functionality, we hoped we would be able to take advantage of the fact that streams come with their own "best practices" workflow. Changes naturally propagate between parent and child, and for the most part the user interface prompts you to merge/copy within these direct relationships, so "driving through hedges" is generally discouraged.

The following is a very simplified example of how a rename action (1->2) can be propagated from a parent stream (A) to its child stream (B).





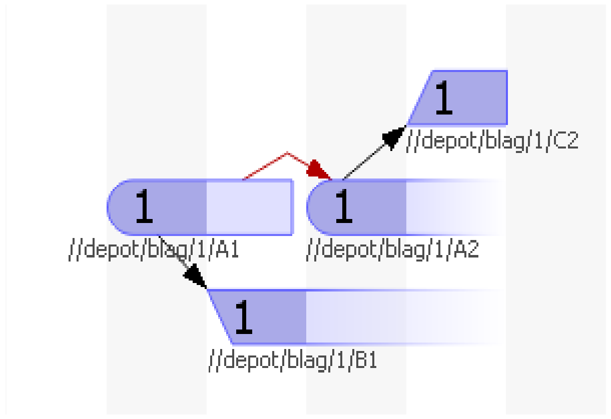
In this example, the branch view maps "A..." to "B...", there is a branch record linking A1 to B1, and there is a move record linking A2 to A1. When we merge A2 through the branch view, all of these relationships come into play. Since B2 (the mapped version of A2) does not exist, we follow the move record back from A2 to A1, and follow the mapping to find B1, which does exist, and has content in common with A2 via the branch record. This tells us that we can integrate A2 to B1 and schedule a filename resolve which will move B1 to B2 (the target of the filename resolve is always the originally mapped path, which corresponds to the source file). Hence:

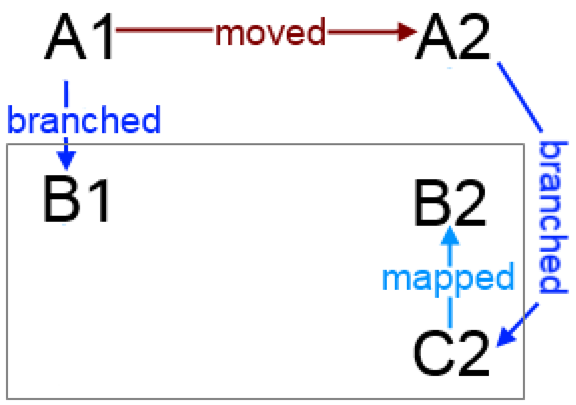
```
//depot/blag/1/B1#1 - integrate from //depot/blag/1/A2#1 (remapped from
//depot/blag/1/B2)
... must resolve content from //depot/blag/1/A2#1
... must resolve move to //depot/blag/1/B2
```

Any changes that have been made to A2 and/or B1 will be merged by the content resolve just as if you'd manually mapped A2 to B1 through a branch view.

Unfortunately, once you step outside parent/child stream relationships this gets more complicated. This can happen because a merge was performed with a manually specified parent that is more distantly related than the original parent, or because a stream was reparented. In either case, if the file names have changed but that information can't be translated through the branch view, it may not be possible to set up the resolves the same way we would for a parent/child merge.

Extending the previous example, suppose A has another child stream, C, that was branched after the rename. Merges between A and C don't have to contend with the rename at all since the paths already match. If C is merged directly to B, though, the differences in file path become problematic.





As in our previous example, the source file (C2) maps to an empty target path (B2), but this time there is no move record in C. The move record still exists in A, but within the context of this merge there is no mapping from A to either B or C, and when we're trying to match filenames to each other, we rely on the mapping to tell us how each move record in the source maps to the target and vice versa. The end result of this merge, then, is to create B2 as a new file, which puts B in the situation of having two variants of the same file, and confusion multiplies from there.

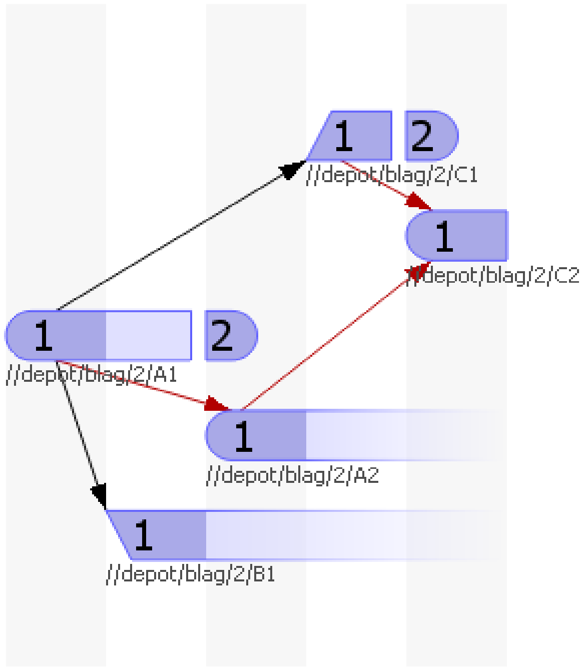
The situation to be careful of, then, is reparenting between streams that are on opposite sides of a refactoring effort – after a bunch of refactoring changes, parent/child merges will get everything lined up correctly, but jumping directly between siblings can be problematic. Here are a couple of ways to get those sibling streams lined up to make merges go more smoothly:

1. Sync the files up by merging from the parent stream first.

This is the most straightforward fix. Using our example, prior to merging C to B (or as part of the reparenting operation), merge A to B (at least as far as the C branch point) to make sure that B has all of A's changes that C does. Once B is synced up with the version of A that C was branched from, merges between C and B will go much smoother.

2. Create branches that are likely to be reparented with deeper history.

This requires foresight and makes branches "heavier", but also provides a lot of flexibility by giving each branch all of the history it needs to be merged with any child of its parent. In our example, we would do this by creating C as a branch from an earlier point on A before copying the latest changes from the desired branch point on A to C. Here's what that looks like:



Now C contains a move record that associates C2 to C1, which allows the rename to be merged into B just as it would be from A:

```
//depot/blag/2/B1#1 - integrate from //depot/blag/2/C2#1 (remapped from
//depot/blag/2/B2)
... must resolve content from //depot/blag/2/C2#1
... must resolve move to //depot/blag/2/B2
```

We are currently experimenting with adding features in the server to make this type of "deep history" easier to create, so that sites which commonly reparent streams after refactoring the mainline will be able to do so and merge changes between the newly reparented streams even when none of the files have the same names at their head revisions.