Subversion 1.9 for Developers

Enterprise Features

Outline

- Branch management
  - When to branch
  - When to merge
- Merging and merge tracking
  - Why merge tracking (by use case)
  - CollabNet merge management GUI
  - Interactive conflict resolution
- Working copy management
  - Sparse checkouts
  - Changelists
  - Peg revisions
  - Externals

Branch management

Know when to branch

- The “Never-Branch” Approach.
  - Users commit their day-to-day work on /trunk.
  - No merging required.
- The “Always-Branch” Approach.
  - Each user creates/works on a private branch for every coding task.
  - When coding is complete, someone merges the changes to a release branch.
  - When a release is complete, someone merges the release branch to /trunk.
- The “Branch Based on Need” Approach.
  - Some branches are used for long term development.
  - When coding is complete, someone merges the changes to /trunk or a release branch (later merging that branch to /trunk).

Know when to merge

BEST PRACTICES

- Merge as often as feasible.
  - Isolation can be useful, but don't let it last too long.
  - Avoid the big hit.
  - The smaller the two ranges of change sets involved, the smaller the chance of conflicts.
- Merge to close out a branch.
- Merge at key milestones.
  - Production releases.
  - Service pack releases.

Merging and merge tracking

Merge tracking – why track merges?

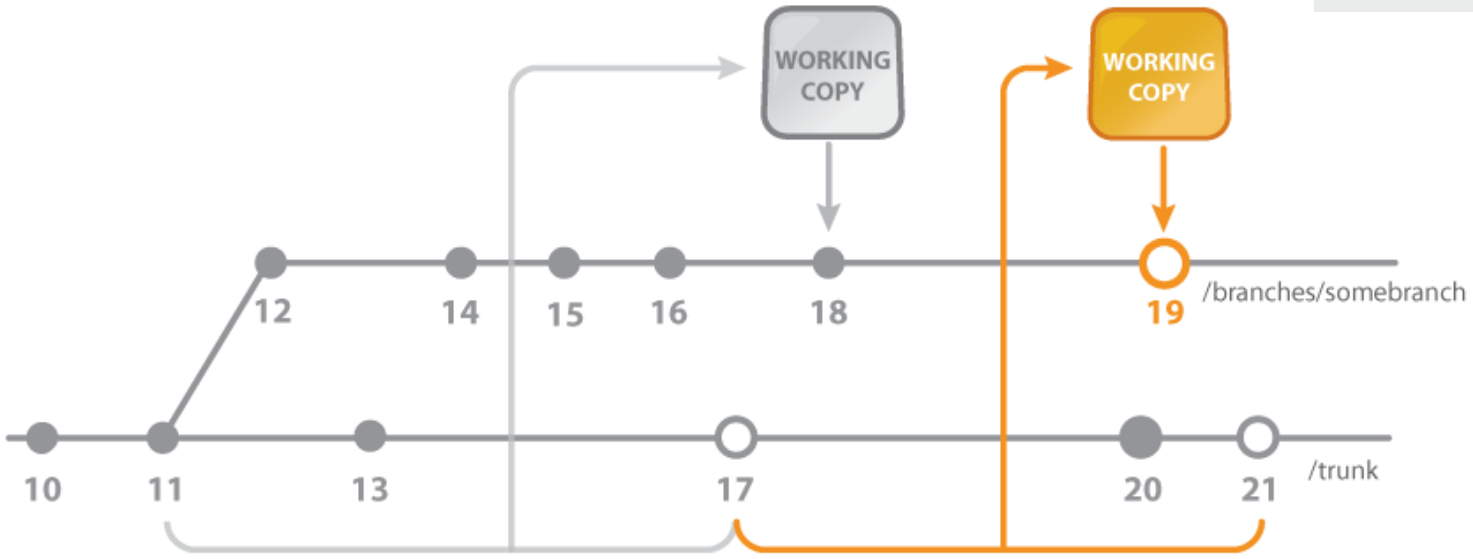
- Automatically handle situations such as duplicate merges.
  - Remembers previous merges.
  - Prevents re-merging.
  - Eliminates missed changes.
- Reduce errors and administration overhead.
  - Teams that merge often will see increased productivity.
- Add auditibility and traceability.
  - What code was merged when and where?
  - Vital to certain industries with high demands on product safety, e.g. automotive, aerospace, medical equipment, etc.
- Facilitate implementing more advanced branching strategies.
  - Get the full advantages of parallel development on multiple branches.

Merge tracking use case – why repeated merge?

Repeated merge:

- Tracks which change sets have been applied where.
- Avoids duplicate merges.
- Provides transitive merge information (next slide).

```
svn:mergeinfo
trunk:11-21
```



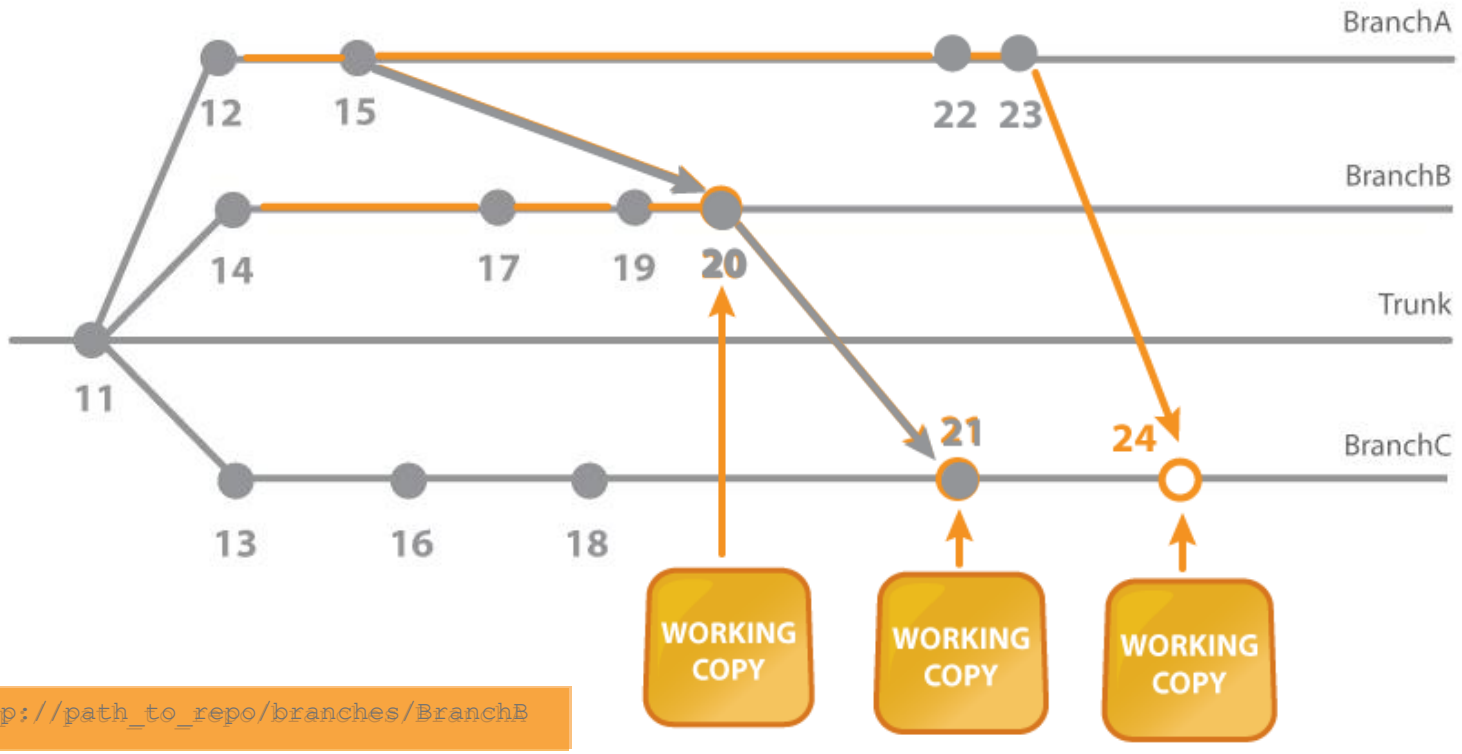

```
svn merge http://path_to_repo/branches/trunk .
```

Merge tracking use case – why transitive merge?

Transitive merge:

- Merges are aware of contributions from other merges.
- Merge records include the contributions to the path being merged from.
- Merging does not revisit changes that were contributed from a previous merge to another branch (duplicate merges).

```
svn:mergeinfo
/branches/BranchA:11-23
/branches/BranchB:11-20
```

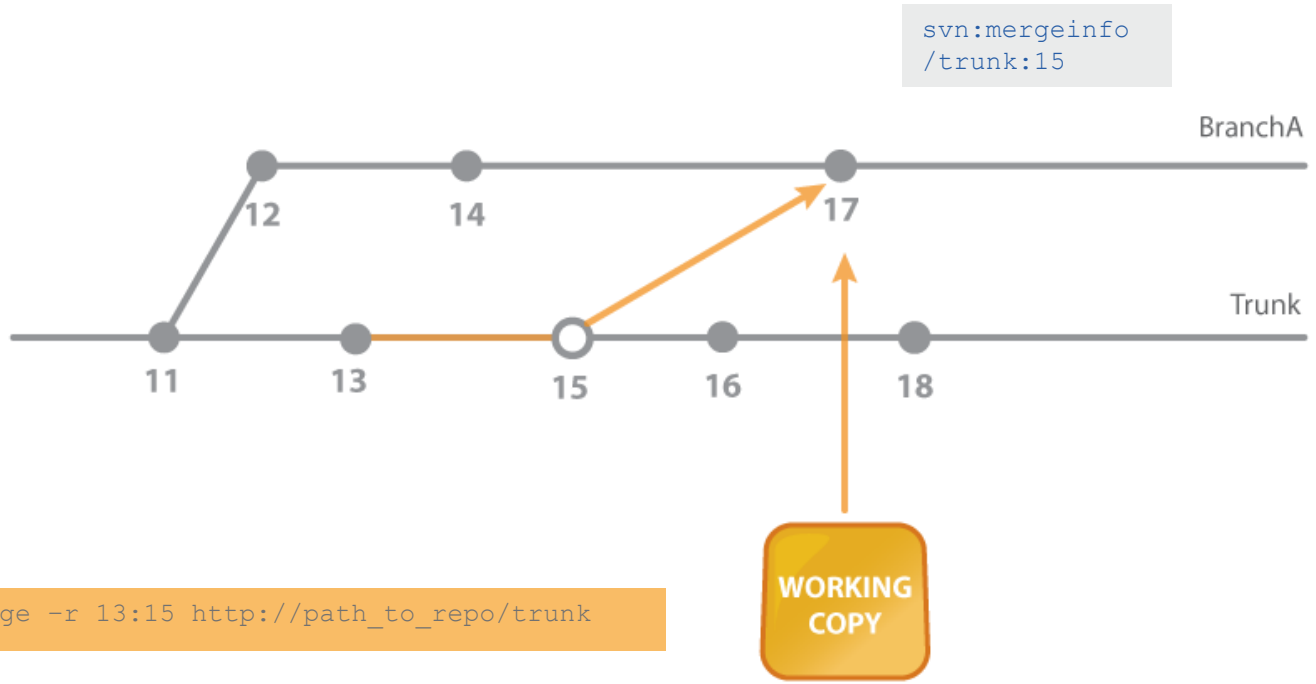



```
svn merge http://path_to_repo/branches/BranchB
```

Merge tracking use case – why cherry picking?

Cherry picking:

- Only applies a selection of revisions, not inclusive of all eligible merge candidates.
- Supports future range merges where the selected revisions are not revisited (no duplicate merges).

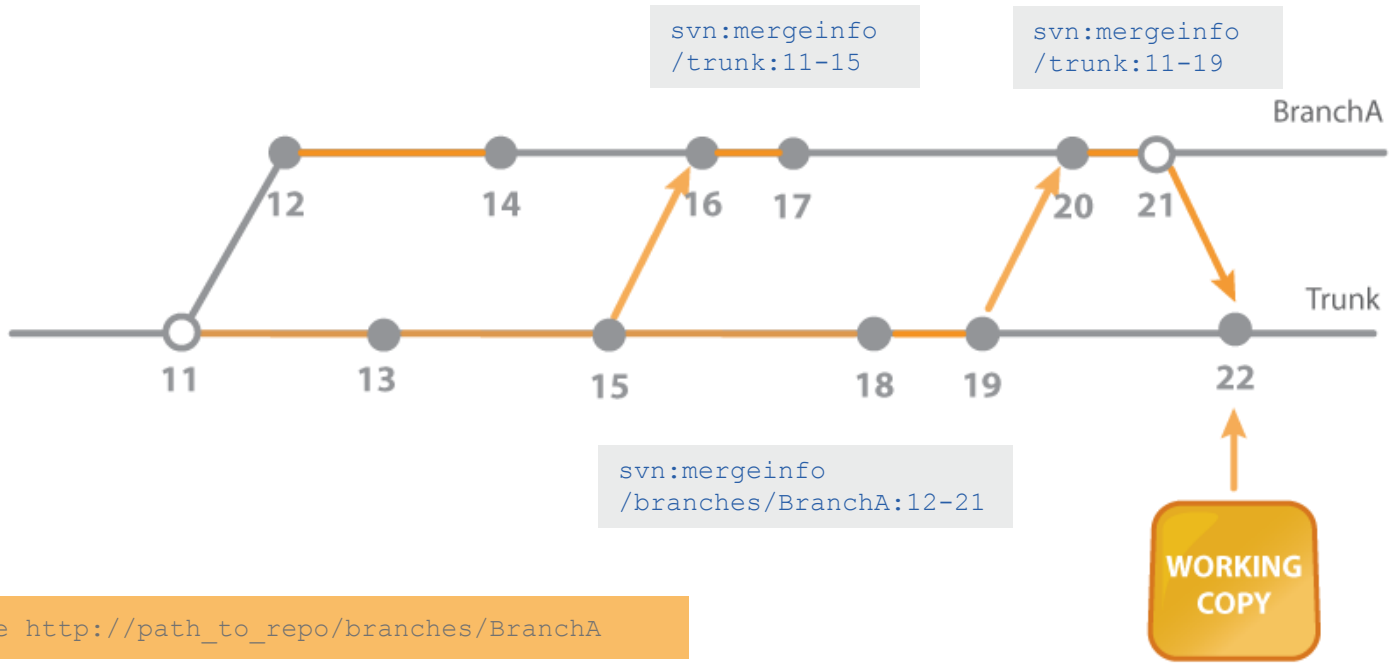



```
svn merge -r 13:15 http://path_to_repo/trunk
```

Merge tracking use case – why child and parent merges?

Child/Parent merges:

- Merge a child branch into its parent branch honoring merge resolutions from the parent to the child (avoids duplicate merges).



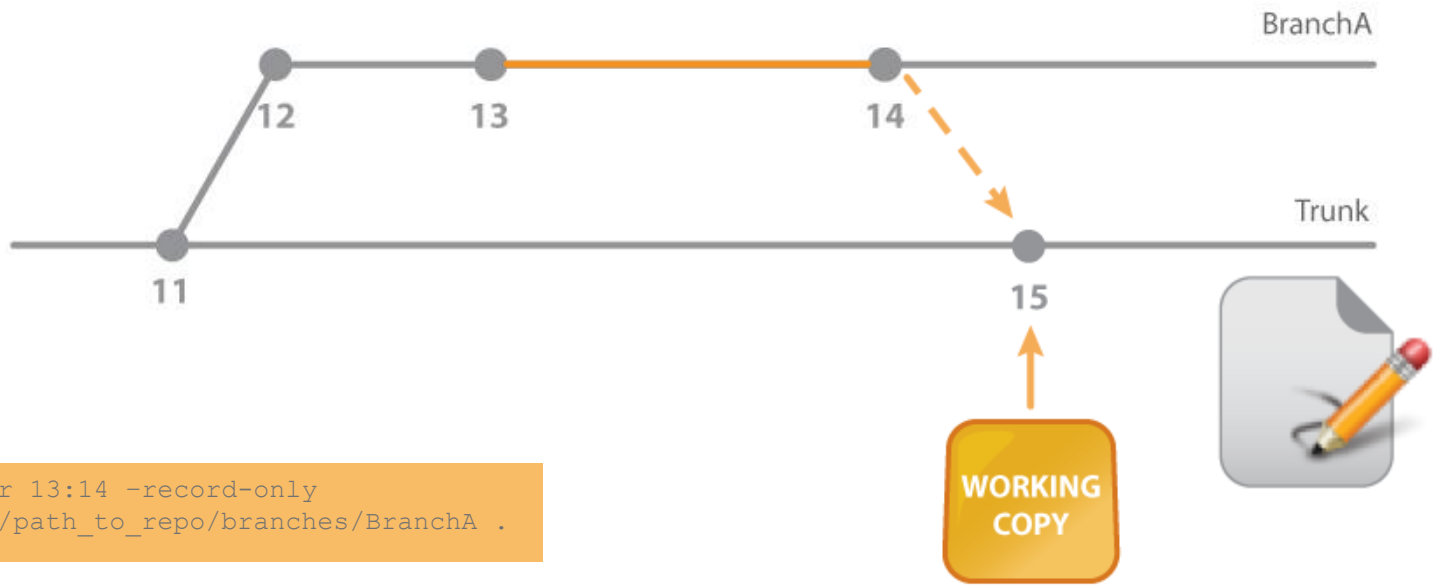

```
svn merge http://path_to_repo/branches/BranchA
```

Merge tracking use case – why manual merge?

Record manual merge:

- Allows change sets to be marked/documented as merged though the merge was done manually by the user.
- Creates a revision block to prevent a revision from being merged to a specific branch.

```
svn:mergeinfo
/branches/BranchA:14
```

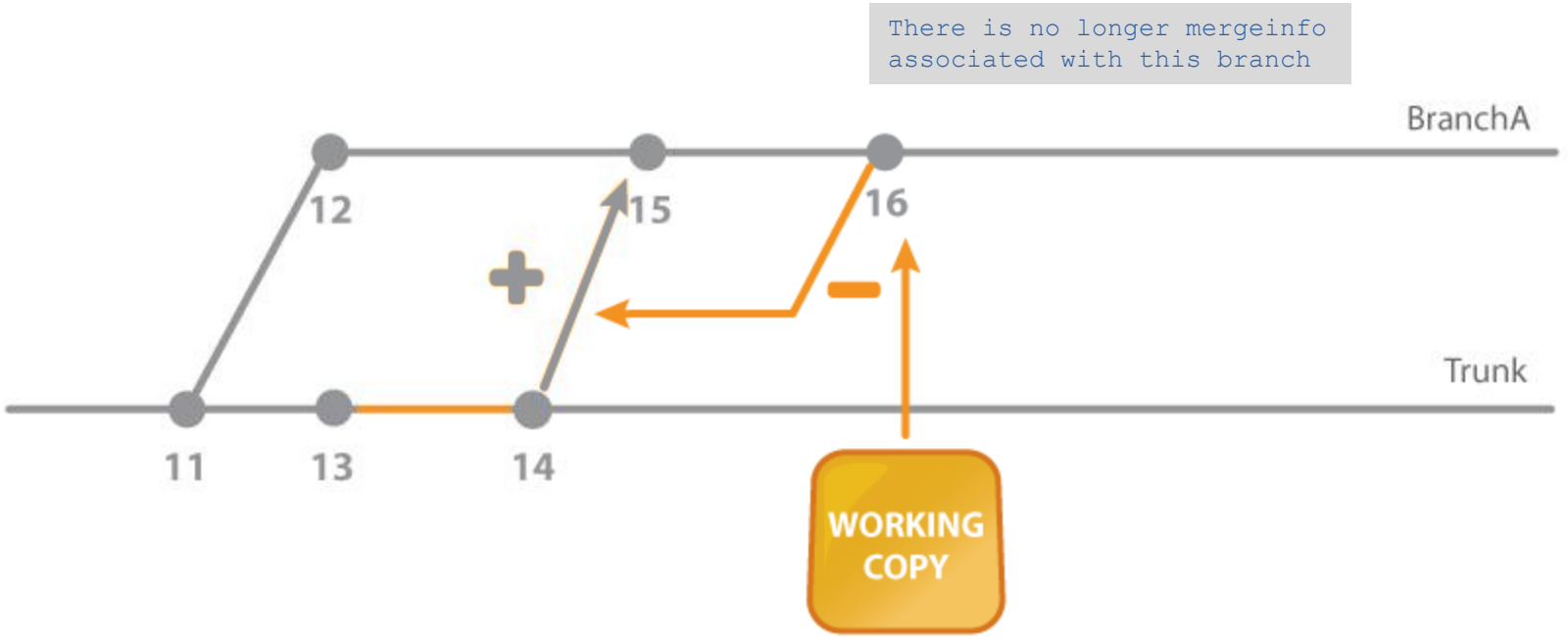



```
svn merge -r 13:14 -record-only
     http://path_to_repo/branches/BranchA .
```

Merge tracking use case – why rollback merge?

Rollback merge:

- Undoes a merge (or any committed change).
- Allows the unmerged change to be remerged later.

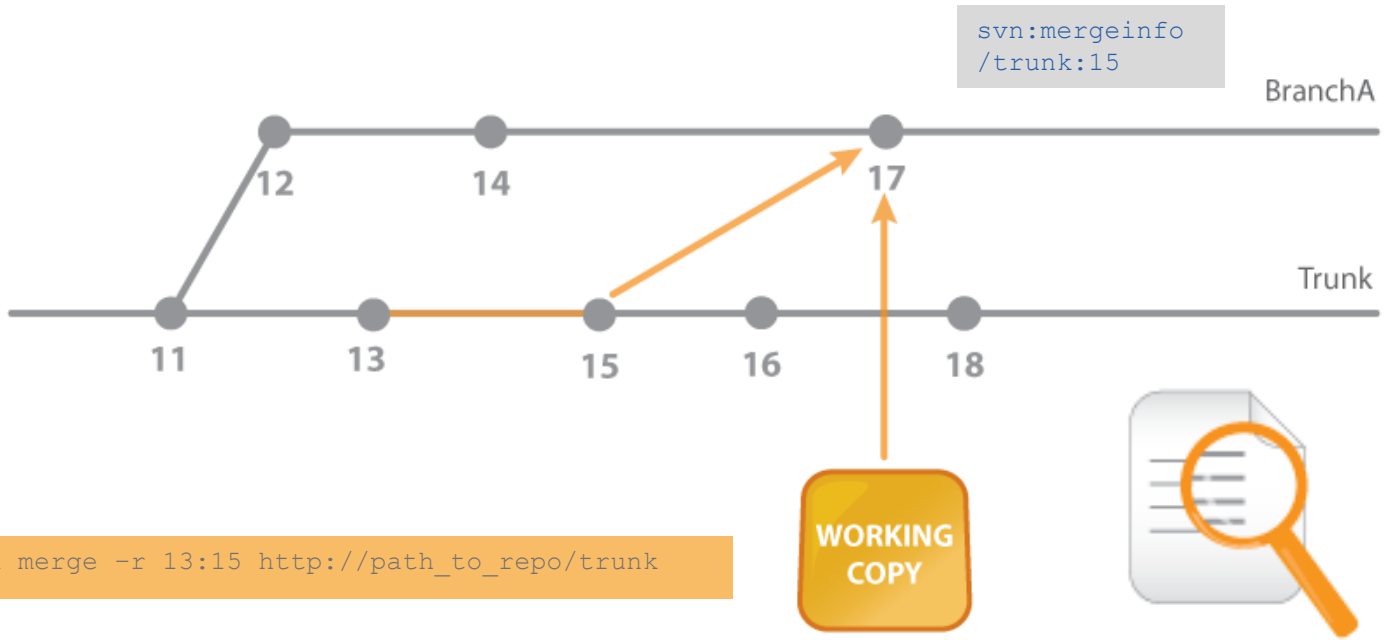



```
svn merge -r 16:15
    http://path_to_repo/branches/BranchA
```

Merge tracking – auditing

Merge auditing:

- Merge data automatically added to svn:mergeinfo.
- Reports merge data using the mergeinfo operation.
- Blame and log operations can report based on this information.



Merge tracking – auditing

- Property (`svn:mergeinfo`) set on files and directories records all merge information.

- Merge history discovery
  - Branches remember their origins.
  - `mergeinfo` operation provides answers to the questions:
    - What changes have I merged into a branch?
    - What changes are eligible for merge into a branch?

- Traceability
  - By recording what revisions exist on what paths, Subversion provides traceability of what made it where.
  - Automation is key to ensure the information is consistent and reliable.
  - Switch `--use-merge-history` (single-character shortcut `-g`).
  - `log -g`: includes what changes were merged by a commit.
  - `blame -g`: shows the original committer and revision for each line in a file.

Merge tracking – CollabNet GUI merge client

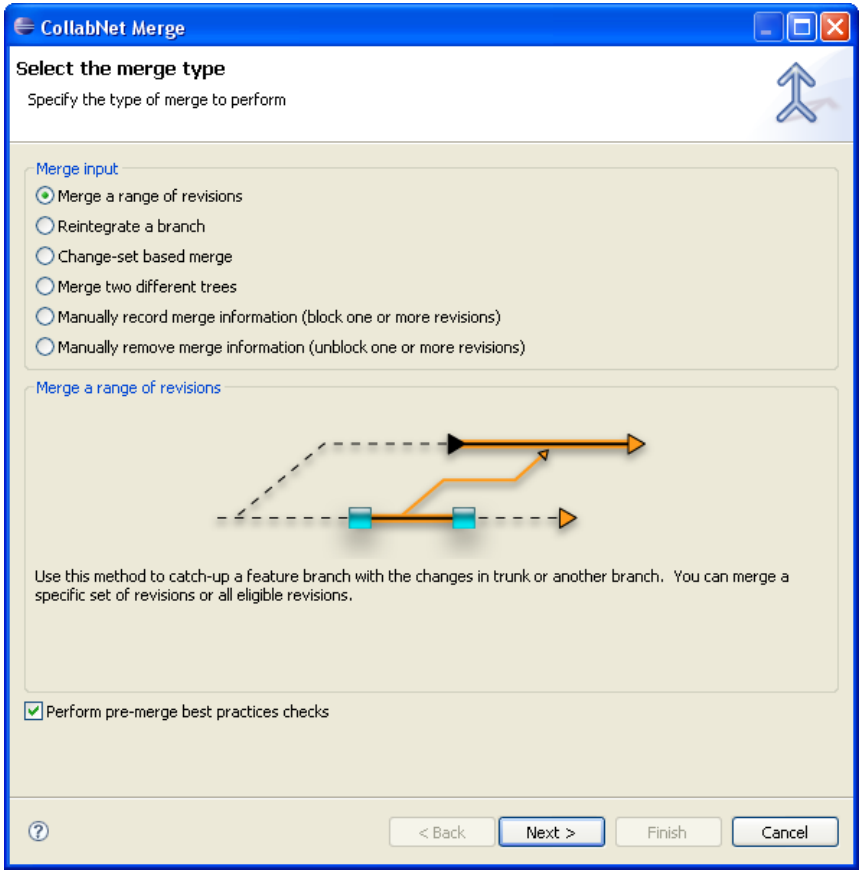
- CollabNet GUI Merge Client for Subversion.
  - The goal of the client is simple: make merging easier.
  - The CollabNet GUI Merge Client is packaged within:
    - CollabNet Desktop - Eclipse Edition.
      - Built on top of Eclipse and Subclipse.
      - Can be used across multiple Eclipse projects (same repository).
    - CollabNet Desktop - Visual Studio Edition.
      - Built on top of Visual Studio and AnkhSVN.
  - Desktops available at: http://www.open.collab.net/downloads/integrations/.

BEST PRACTICE

- Before performing a merge best practices can be considered:
  - It is a best practice to not have any uncommitted changes in your working copy.
  - Your working copy must be pointing to the area of the repository that you want the merge results to be commited to.
  - You should update the entire working copy to HEAD before beginning the merge.
  - You shouldn't have switched children in your working copy.

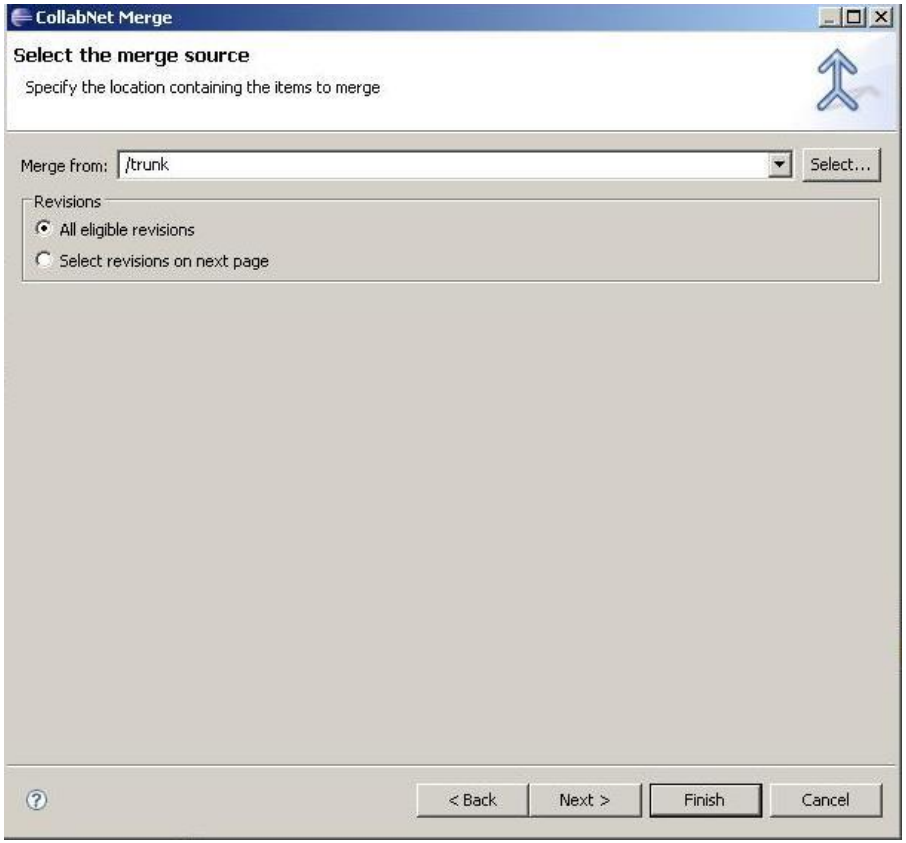
CollabNet GUI merge client – Select merge type

- Select the merge type
  - Specify the type of merge you want to perform.
  - Visually confirm the merge type by the associated diagram.
  - Evaluate using the option to perform pre-merge best practice checks and provide warnings.



CollabNet GUI merge client – Select merge source

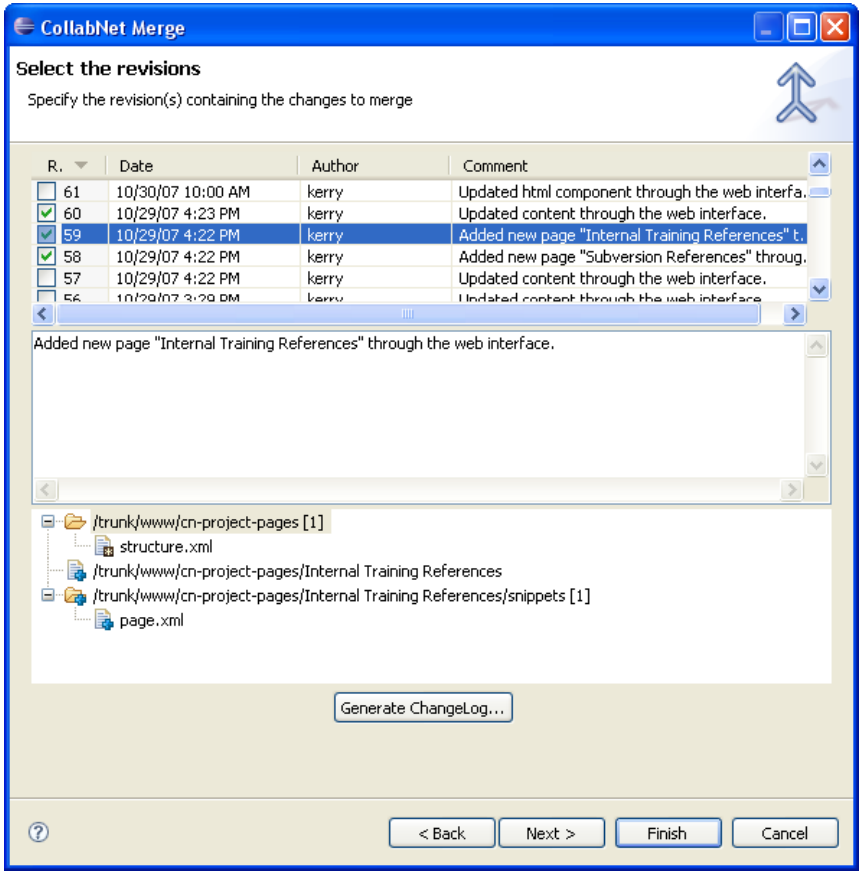
- Select the merge source
  - Indicate from where you want to merge.
  - Defaults to the branch's parent (i.e., where it was created from).
  - Choose either all eligible (unmerged) revisions or select specific revisions to merge
  - Completes pre-merge checks successfully before displaying this dialog.



Example: Range of Revisions

CollabNet GUI merge client – Select revisions

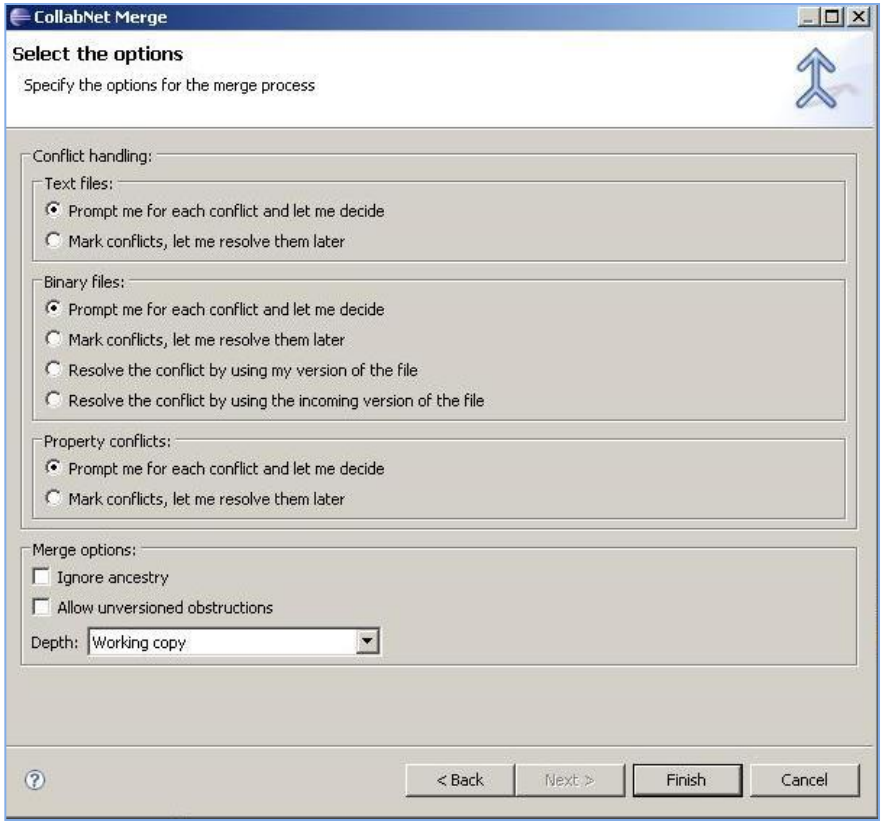
- Select the revisions (if chosen):
  - This page will only list the revisions that have not already been merged or blocked (i.e., marked manually as merged).
  - You can cherry pick revisions to be merged.
  - You can see what revision had changes, when it was committed, by whom and what they said.
  - You can see what was changed (in a structural format) for each of the candidate revisions.



Example: Range of Revisions – Cherry Picking

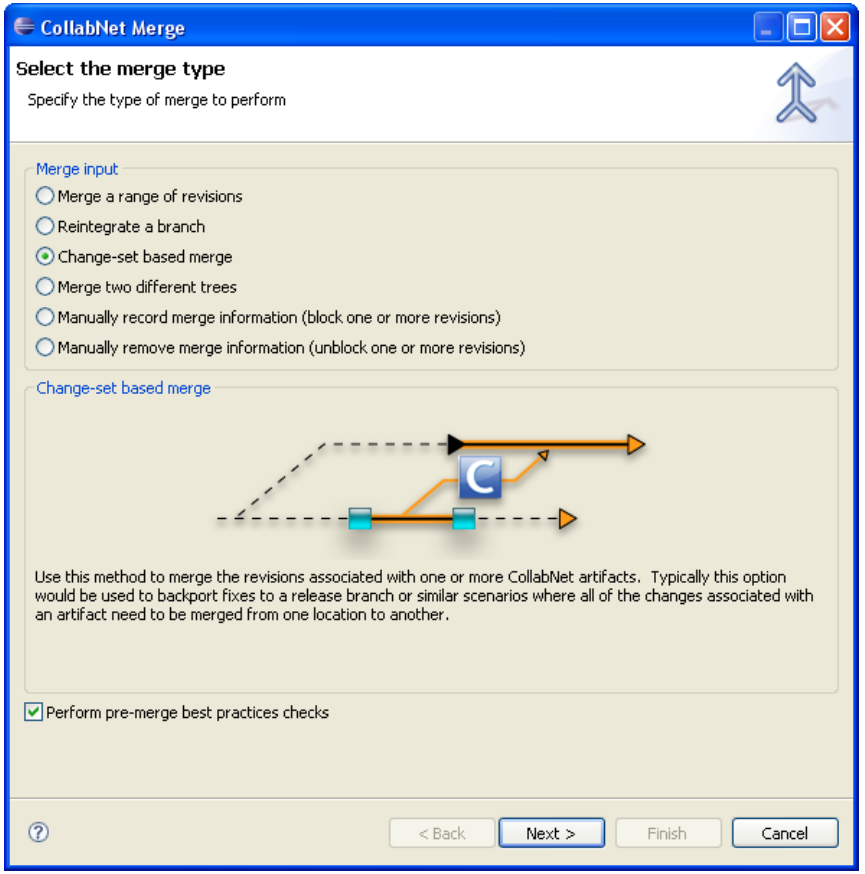
CollabNet GUI merge client – Select merge options

- Select merge options
  - This dialog is standard for all merges.
  - You can decide whether to have interactive conflict resolution on text files or defer resolution.
  - You can determine how to handle binary files that are in conflict.
  - You can define how to handle property values that are in conflict.
  - You can select standard merge options including the depth you want to go in your working copy.



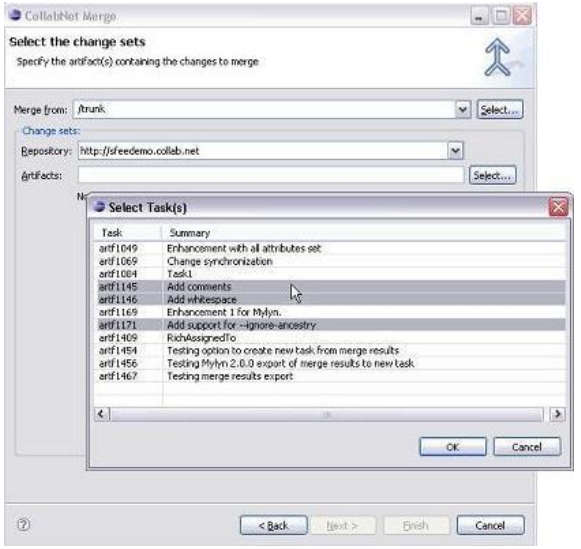
CollabNet GUI merge client – Change-set based merge

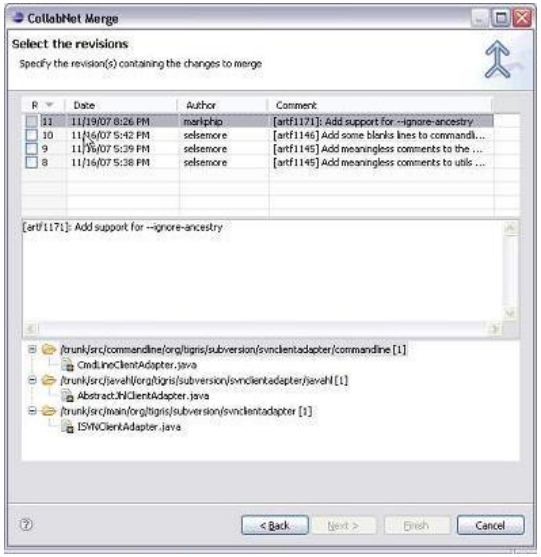
- Change-set merge:
  - Only for CollabNet TeamForge users that also have chosen to associate commits with tracker artifacts.
  - Select what to merge based on the association between commits and tracker artifacts.
- Benefits:
  - Merging all changes related to a specific tracker artifact (e.g.: a bug fix or new feature) is a more natural workflow.
  - One more level of traceability is achieved.
  - Integrated tools, processes and workflow makes the team more productive.



CollabNet GUI merge client – Change-set based merge

- Select the change sets:
  - Allows manual entry of artifacts or the ability to select the desired ones using a query.
- Select the revisions:
  - Allows you to see the Subversion revisions that are associated with the selected artifacts.
  - Provides you the ability to selectively choose which revisions to merge using information on what changes were in the revision and what was said about the commit.





Interactive conflict resolution

- An interactive response to conflicts encountered during the following operations:
  - Merge
  - Update
  - Switch
- Configurable behavior:
  - Pre-specify directives like “always use the version from my merge source”,
  - Selectively disable by using the `--non-interactive` option,
  - … or disable permanently by a setting in your run-time config file:
    `'[miscellany] interactive-conflicts = no'`
- `resolve` command:
  - Allows you to interactively resolve conflicts later if you postponed them earlier.
  - `--accept` option:
    - Lets you mark a conflict as resolved and resolve it by choosing a specific version of a file (i.e., working copy or repository).
    - Useful in resolving binary file conflicts.

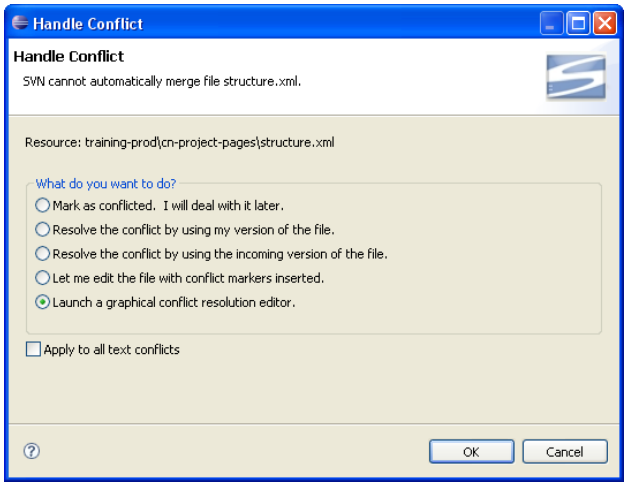
Interactive conflict resolution command line

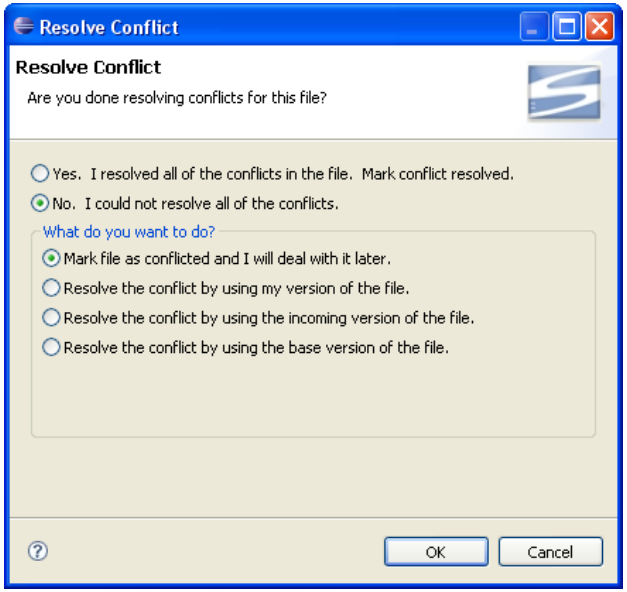
- Command line opportunities to resolve conflicts:

```
$ svn up contrib/client-side/svnmerge_test.py
Conflict discovered in 'contrib/client-side/svnmerge.py'.
    (e) edit - change merged file in an editor
    (df) diff-full - show all changes made to merged file
    (r) resolved - accept merged version of file
    (dc) display-conflict – show all conflicts (ignoring merged version)
    (mc) mine-conflict – accept my version for all conflicts (same)
    (tc) theirs-conflict – accept their version for all conflicts (same)
    (mf) mine-full - accept my version of entire file
    (tf) theirs-full - accept their version of entire file
    (p) postpone - mark the conflict to be resolved later
    (l) launch - use external tool to resolve conflict
    (h) help - show this list
Select: (p) postpone, (mf) mine-full, (tf) theirs-full,
        (s) show all options:
```

Interactive conflict resolution GUI example

- Several options available when using a GUI client:
  - Deal with the conflict later.
  - Select a specific version of the file.
  - Resolve the conflicts manually.
  - Launch a graphical conflict resolution editor.
- Option to apply to all text conflicts in the current merge.
- Finally, tell the client if you actually resolved the conflicts.





Working copy management

Sparse checkouts

- Sparse checkouts are client-side, per-working-copy.
  - Server has no record of a working copy's structure (it can be made aware of what an update should cover).
  - There is no "profile" to create a similar working copy (though commands can be scripted to do so).
- Selected depth is "sticky" – maintained across operations.
- Different depths can be mixed in one working copy.
- Like any client side functionality, this doesn't require a 1.7 or later server to utilize the feature:
  - Legacy servers will send back information the client doesn't want.
  - The client will just ignore it - it's slow, but correct.

Sparse checkouts (Cont'd)

- `--depth` option:
  - Affected subcommands: `update`, `status`, `info`, `switch` and `checkout`.
  - Sets depth values as it updates the working copy.
  - Obsoletes the `--recursive` (`-R`) and `--non-recursive` (`-N`) options.
  - Values:
    - **infinity** *(default)*: go as deep as exists.
    - **empty**: updates will only pull in files or subdirectories already present and checkouts will pull in an empty directory.
    - **files**: updates will pull in any files not already present, but not subdirectories.
    - **immediates**: gets a "high-level overview" tree (top level files and empty subdirectories).
    - **exclude**: don't include this path until and unless told to do so.
- Merging a sparsely populated directory (not a best practice):
  - Non-inheritable `svn:mergeinfo` is set on the deepest directories present:
    - Directories with depth == **empty**: The directory gets non-inheritable `svn:mergeinfo`.
    - Directories with depth == **files**: The directory gets non-inheritable `svn:mergeinfo`. Any child files present get inheritable `svn:mergeinfo`.
    - Directories with depth == **immediates**: The directory and any child files present get inheritable `svn:mergeinfo`. Any directory children present get non-inheritable `svn:mergeinfo`.

Sparse checkouts – example usage

- Checkout an empty tree creating an empty working copy:

```
svn co --depth=empty http://.../A Awc
```

- Checkout only the top-level files and no subdirectories:

```
svn co --depth=files http://.../A Awc1
```

- Checkout only the top-level files and empty subdirectories (not sure what sub trees you need, only want to pull them in as needed):

```
svn co --depth=immediates http://.../A Awc2
```

- Default depth is the same as pre-1.5 - infinity:

```
svn co http://.../A
```

Sparse checkouts – example usage (Cont'd)

- Selective update still available (just updates the working copy respecting any depth settings):

```
svn up Awc
```

- Pull in a sub tree with depth infinity:

```
svn up Awc/B
```

- Pull in a sub tree with depth immediates:

```
svn up –depth=immediates Awc/D
```

- Pull in a couple of files in an empty sub tree:

```
svn up Awc/D/sub1/foo.c Awc/D/sub1/bar.h
```

- Remove everything in a subdirectory and the subdirectory itself:

```
svn up --depth=exclude Awc/D
```

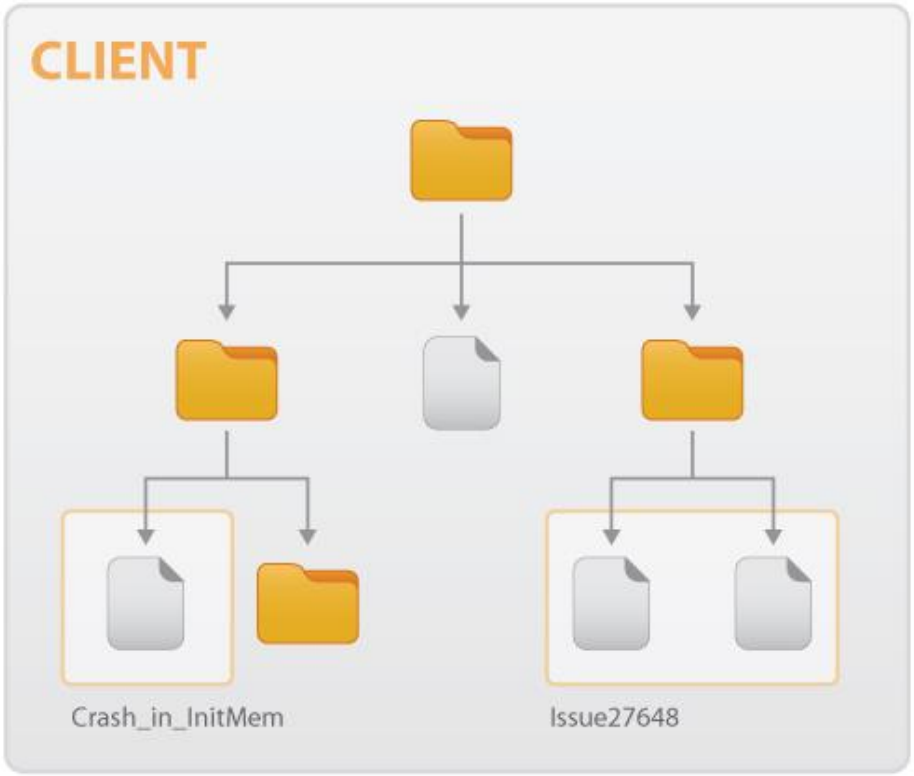
Changelists

- A changelist associates multiple files together.
- Changelists are arbitrary labels applied to working copy files:
  - Users are allowed to invent their own names.
  - They are client-side, per-working-copy.
- Changelists are a way to separate and organize logical changes:
  - Users create, view, and manipulate sets of files in a working copy by referring to them by the changelist name.
- The `changelist` command allows a user to define a changelist with an arbitrary UTF-8 name, as well as add member paths.

```
$ svn changelist MYCHANGE foo.c bar.c
Path 'foo.c' is now part of changelist 'MYCHANGE'.
Path 'bar.c' is now part of changelist 'MYCHANGE'.
```

Changelists - usage

- A user working on more than one set of logical changes at a time.
  - Reduce context-switching costs…
  - But use with caution: working on multiple changelists in a working copy includes the risk of committing something that doesn't build.



Changelists (Cont'd)

- Changelists allow you to:
  - Group a subset of local changes, and
  - Run subcommands on those logical groups.
- Notes:
  - A file can be in at most one changelist – no overlapping.
  - A file being put into another changelist implicitly removes it from its current changelist.
  - Directories are not allowed in changelists – helps prevent overlaps.
  - Once you commit, changelists disappear unless you indicate they should be retained (i.e., use the commit command option `--keep-changelists`).

Changelists – Supported Operations

- Define a changelist by explicitly adding/removing paths to it (`changelist`).
- See all existing changelist names and their member paths (`status`).
- Destroy a changelist definition all at once (`changelist -R --remove --changelist` *`changelistname`*).
- Examine all edits within a changelist (`diff`).
- Revert all edits within a changelist (`revert`).
- Receive server changes only from paths within a changelist (`update`).
- Commit all edits within a changelist (`commit`).
- Fetch or set props on every path within a changelist (`proplist/propset/propedit/propget/propdel`).
- Continue using a changelist after a commit (`commit --keep-changelist`).

Peg revisions

- A peg revision is a notation for a particular path at a particular revision.
- A peg revision looks first at the revision and then for the path.
- Many times, a peg revision resolves to the same path and revision as the –r argument (which looks at the path and then finds the revision).
  - i.e., -r 12 foo.c may be equivalent to foo.c@12
- A peg revision can resolve to a different path and revision when objects are deleted or moved and other objects created with the same name. For example:
  - Adding foo.c to Subversion and committing creates revision 101.
  - Deleting foo.c from Subversion and committing creates revision 102.
  - Adding foo.c to Subversion and committing creates revision 103.
  - svn log –r101 foo.c returns nothing (i.e., fails as the current foo.c didn't exist in revision 101).
  - svn log foo.c@101 returns history of the first foo.c file.

Externals details

- The URLs may include peg specifications:
  - The format has the revision selector (if not HEAD or designated by peg notation) first, followed by the URL and then the working copy path the external is checked out or exported into.

```
        http://example.com/repos/zig      foo1
-r 1234 http://example.com/repos/zag      foo/bar1
        ../../component1/zig@HEAD         foo2
        ../../component2/zag@1234         foo/bar2
```

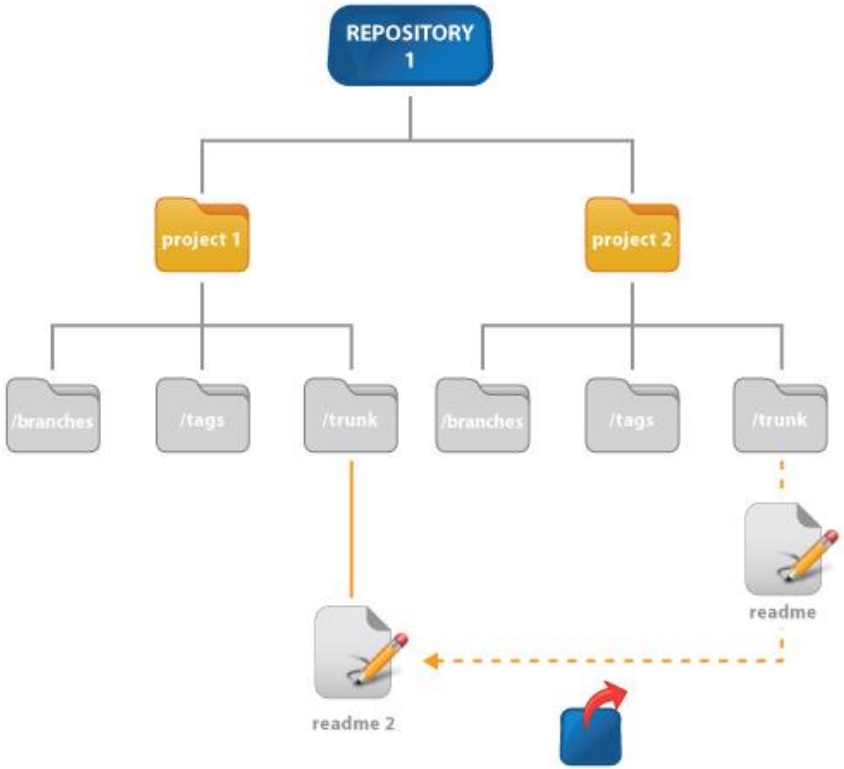
  - Peg specifications are allowed but not required.

- The URLs in an svn:externals specification can be relative.
  - Four different relative externals are supported:
    - ../: relative to the directory with the svn:external property
    - ^/: relative to the repository root
    - //: relative to the scheme
    - /: server root relative URLs
  - When Subversion sees an svn:externals without an absolute URL, it takes the first argument (with the exception of a revision selector) as a relative URL and the second as the target directory.

Externals example 1

- Map a file in one project to a working copy file in another project within the same repository.
  - On https://repo1/PROJECT1/trunk set the svn:externals property

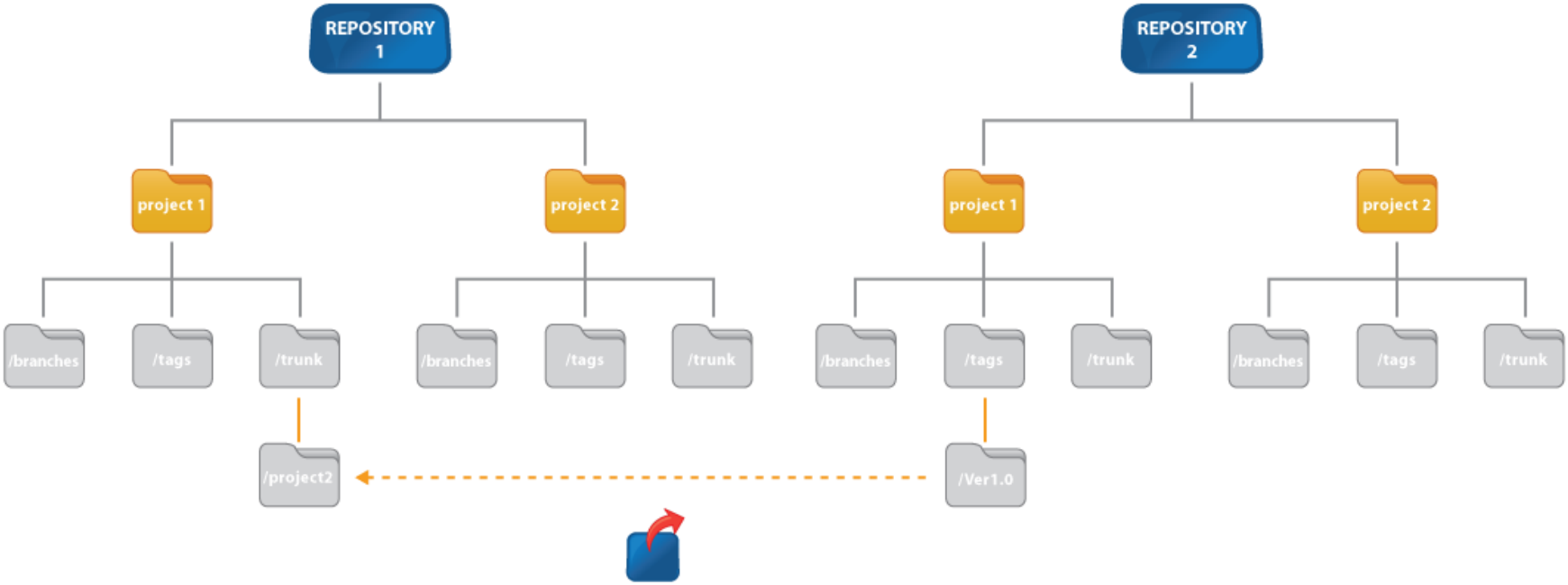
```
svn propset svn:externals "../../PROJECT2/trunk/readme@7654 readme2" .
```



Externals example 2

- Map directory in one project into a working copy directory referencing a project in a second repository.
  - On https://repo1/PROJECT1/trunk set the svn:externals property

```
svn propset svn:externals "https://repo2/PROJECT1/tags/ver1.0 project2" .
```



Thank You

About CollabNet

CollabNet is a leading provider of Enterprise Cloud Development and Agile ALM products and services for software-driven organizations. With more than 10,000 global customers, the company provides a suite of platforms and services to address three major trends disrupting the software industry: Agile, DevOps and hybrid cloud development. Its CloudForge™ development-Platform-as-a-Service (dPaaS) enables cloud development through a flexible platform that is team friendly, enterprise ready and integrated to support leading third party tools. The CollabNet TeamForge® ALM, ScrumWorks® Pro project management and SubversionEdge source code management platforms can be deployed separately or together, in the cloud or on-premise. CollabNet complements its technical offerings with industry leading consulting and training services for Agile and cloud development transformations. Many CollabNet customers improve productivity by as much as 70 percent, while reducing costs by 80 percent.

For more information, please visit www.collab.net.

CollabNet, Inc.
8000 Marina Blvd., Suite 600
Brisbane, CA 94005

www.collab.net

+1-650-228-2500
+1-888-778-9793
blogs.collab.net
twitter.com/collabnet
www.facebook.com/collabnet
www.linkedin.com/company/collabnet-inc