CECS-343 — SWEngr Intro — VCS Project 3

VCS Project 3— Merge

Introduction

This is the third part of our VCS. In this project part, we add the ability to merge two project tree snapshots (that are based on the same repo, and hence the same development project). Note that we should already have a natural branching effect due to check-out (of a project snapshot, AKA the Kid). Because the Kid checkout's command line includes the source manifest snapshot (or its label), we know the Mom snapshot from which the Kid was copied. We also should know for every manifest, the project tree folder it represents (the source folder on Create or Checked-in, and the target folder on Checked-out). These manifest command line features allows us to trace a given project folder's repo snapshot all the way to the repo's original Create command. This is to say, the set of snapshots in the repo for a DAG (Directed Acyclic Graph) where each node is a snapshot.

The Merge ability is typically used to merge two snapshots that have different modifications (usually in different folders/files), or to merge a branch snapshot of modifications (e.g., a bug fix, or a new feature) back into a project mainline's “head” snapshot. (The “head” is the latest snapshot along any branch in the repo.)

We will do the merge in two parts using two commands: **Merge-out** and **Merge-in**. The M**erge-out command** will gather the information needed for the user to manually do a 3-way merge. (Sorry, user, but this is still a somewhat no-frills VCS.) Next, we presume that the user will do the necessary merging by hand. (While there are 3-way merge tools, they all expect the user to do the hard part of deciding how to merge to files with overlapping changes.) The **Merge-in command** will complete the merge by "checking in" a snapshot of the user's fully-merged project tree folder.

Also, note that our VCS is project-based, not file-based. We do not run a VCS command to copy an individual file to/from the repo, but instead only to copy an entire project tree folder and its sub-folders and files (which resides outside the repo directory) to/from the repo (where “to repo” is taking a snapshot and “from repo” is recreating a new project tree from a snapshot).

Source & Target

The **Merge-out** arguments are the repo, a repo source snapshot, and a target project tree folder. The source snapshot is called the repo 'R' snapshot. (Note, using the MVC pattern, you should have each major command be a function in your Model; and the UI (the 'VC' in MVC) takes the user's inputs and in turn calls its corresponding 'M' function with its arguments. MVC allows you to change the UI without changing the Model.)

The first action the **Merge-out** does is to automatically check-in that target folder. The target snapshot is called the repo 'T' snapshot. The **Merge-out** will (maybe) add new files/folders from the R snapshot to the T project tree, with special file naming.

Merge File Non-Collisions

For R snapshot path/files that don't exist in the target project tree, the **Merge-out** command will copy them, creating new folder paths as needed.

Merge File Collisions

For an R snapshot path/file that exists in the target project tree but the R file is identical (via artifact ID) with its target file, the **Merge-out** command can skip copying that R file into the target tree.

When an R file and its T file both exist but **don't match** (don't have the same artifact ID) as their corresponding target files, they are called **collision files**. Because we are merging an entire project tree, there may be many collision files for one merge-out command. When we detect that one of the many project snapshot files collides (has a different artifact ID) with the corresponding file of the other snapshot, we will add the R file to the target project tree, and we will give a special suffix to each files (identical) name in order to distinguish them. The mismatching R file from the repo source snapshot will be added, but with a suffix of "_MR" meaning "Mismatch-from-Repo". The corresponding T file in the target project tree will have its filename suffixed with "_MT" meaning "Mismatch-in-Target". Both files will retain their existing extensions (e.g, "fred_MR.java" and "fred_MT.java").

Also, a third corresponding file, Grandma file will be added to the target project tree; and will have its filename suffixed with "_MG" meaning "Mismatch-Gramdma".

Note that all three (R, T, and G) files will have the same path and original filename. Recall that the path can be checked with the P part of the artifact ID.

Also, the merge-out command will create a manifest file which includes the command and arguments, and the name of the "grandma" snapshot manifest file and all the files that end up in the target project tree, and an indication flag for each collision file involved on that file's line in the merge-out manifest.

Grandma Manifest

You will need to find the "grandma" snapshot of the source and target snapshots. This is easy because of the DAG nature of the mom-kid snapshot pairs. The grandma snapshot is the **most recent common ancestor** snapshot of the R and T snapshots. The "grandma" snapshot is the same for all pairs of colliding files. That Grandma 'G' snapshot is determined by the path of immediate ancestor snapshots from a given (R or T) snapshot back to the root of the repo snapshot DAG.

To find the G manifest, you can walk the DAG back to the root from both the R manifest and the T manifest. (There may be more than one path backward for each.) Each walk back to the Root produces a path from the root to that manifest Leaf. You can look at the common prefix between an R path from the root and a T path from the root. The G manifest file is **the last manifest** in **the longest common prefix**. (For purposes of this project, we will assume that no prior Merge commands have been issued – no frills. Hence, the DAG will be a normal Tree, and there will be only one Root path to R and to T; thus making it easy to determine Grandma.)

Merge-In

The **Merge-in** command assumes 1) that the user has finished manually fully-merging the R file changes into the corresponding T collision file, for all the collision file pairs, as needed. This includes removing the MR and MG files and removing the "_MT" file suffix, leaving only the merged T file for that file collision. And 2) the user has done no other repo commands since the merge-out command on this project tree. (Commands on other project trees for the same team project can be done, and should not cause a problem.) For example, the 3 "fred_M*.java" files will be used to create a merged T file, "fred.java" file (without the _MT), and the _MR, and _MG files will be removed – all by the user.

Merge-in is merely a duplicate of the check-in command, except that it has a different command name, and because no other commands have been run on the project tree, the **Merge-in** command will always appear as the kid of its mom's **Merge-out** command in the repo's set of manifest files. The merge-in manifest contents are equivalent to the check-in manifest file.

DAG (This section is optional)

Because the mom-kid (parent-child) relationships among the manifest files can be a DAG, finding the grandma snapshot (the most recent common ancestor) of the source and the target is a bit more complicated than finding the common ancestor in a tree.

If, in searching upward from the source or target snapshot you encounter a merge-out and merge-in pair of snapshots, you will have to follow both parent branches upward toward the root (the create-repo snapshot) because the grandma could be in either one of them. A simple way of doing this is to pick one branch to do immediately and put the other mom-branch's snapshot (or whatever is needed to identify and use it later) on a "pending" list (or array, stack, etc.). You will find a grandma candidate snapshot along every path from your (R or T) snapshot up to the root (create-repo snapshot). It is only the most recent of them that is the actual grandma snapshot for this R-and-T merge.

Dot-Files and DAG tracking

If you find it convenient, you can put VCS files (eg, manifests) in the user's project tree at its root folder. If you do so, these files must have a filename beginning with a dot/period, '.', and the prefix “vcsx” for our VCS system. If you decide to do this, make sure that dot-files are **not included** when you check in a snapshot. The “user manual” will tell the user to touch these dot-files at peril!

Team

The team size and naming style is the same as before, but you can change team members from the previous project if you wish.

Testing

Be sure to provide cases that test for at least two conflict-files, each resulting in an MT, MR, and MG triple of files, and at least one non-conflict file where the R and T files are identical, and at least one non-conflict file where one of them doesn't exist. (Optional: You can provide at least one test case where one of the T or R snapshots has multiple root paths.)

Include, in your submitted .zip file, a sample "run" for each test, consisting of directory listings of the project tree and the repo, and of the new manifest file involved. These can be cut-and-pasted into a .txt file for the run.

Technical Debt

This is a class in S/W Engineering. In spite of that, we emphasize working S/W (Rule #0). However, getting to working S/W fast often leaves technical debt – ugly code that is both hard to understand and complicates future modifications. Technical debt will rapidly turn into a Bad Smell if left too long to fester. Therefore, as this is the last rapid delivery, **the technical debt must be paid**, approximately in full; which is to say that your team's source code should be reasonably clean and well-documented.

Academic Rules

Correctly and properly attribute all third party material and references, lest points be taken off.

Project Reports, Submission & Readme, Grading

Same as before.