362 — SWE Fundamentals — VCS Project 3

VCS Project 3— Merged

Introduction

This is the third part of our VCS (Version Control System) project. In this project part, we add the ability to merge two project trees (that are based on the same repo). Note that we already have a natural branching effect due to check-out (of a checked-in project version into a new project tree, AKA the Kid project version) coupled with the fact that we track that project version's parent (AKA the Mom version) ***as identified in the Kid's repo manifest***.

The merge ability helps the user to merge a (repo) project tree (called the 'R' version) that is already in the repo (as represented by a given manifest file) into a given target project tree (called the 'T' version) outside the repo. To ensure that the T version also has an up-to-date manifest, you will first automatically check that version in prior to the "merge".

For example, Fred can merge Jack's project changes that are in the repo (the given R version, checked-in earlier from Jack's project tree) into Fred's current project tree (the given T version). If the merge succeeds (standard merge software is only able to handle simple file differences, but this is usually the case) then Fred can quickly run some tests (Fred's unit tests and the project's regression test suite) on the merged resulting project tree and check-in his resulting merged project tree for others to use. This is how a branch is normally merged back into mainline (although, sometimes this needs SCCB authorization).

If there is a file in the R version that is different in the T version, we will arrange for 3 copies of that file to be in the T version, each with a different name suffix: the R version, the T version and the G version (which represents the most recent common ancestor project version from which both the R and the T versions were derived). We will leave it to the user to do the actual 3-way merge.

Team

The team may contain up to 4 members, and may be different from prior project teams. Pick a three-letter name for the team as described for project #2.

Merge Interface

The user interface for the merge operation is simple. (Note, one should always build a simple command line version of the UI so as to help maintain an MVC/MVP/MVVM architecture, even if the end goal is to have a GUI.) For the merge command, the user needs to indicate the R project version by specifying its repo manifest file, for example by label or by filename. Also, the user needs to indicate the T project version by a path to its root folder. Note, be sure to allow the R manifest to be selected by a label, as well.

Merge Common Ancestor

The two project versions, R and T, to be merged were children of some prior parent versions – R's Mom and T's Mom. (Note that it's possible that their relationship is weird: R is T's Mom, or R is actually the repo root version via the Create command, or R is a deep ancestor of T, but these are simplified versions of the problem.) Thus, w.l.o.g. we will assume that R and T are on different paths in the repo tree of branches. Therefore, there is a common path from the repo root that eventually diverges, one path leading to R and the other path leading to T. And therefore there is a most recent common ancestor (called 'G' for Grandma) for those two paths, that both paths share.

The Merge must identify the G project version, because its files will be used if a 3-way merge is

needed (if R's file version and T's file version are different which you can detect by their artifact IDs).

Merge Results

After the merge command has run, the target (T) project tree (which is outside the repo) should have a single file version with its normal filename (either from R or from T) if that file has no differences in the R tree and the T tree -- either the file only exists in one tree, or it is in both but they have the same artifact ID and hence are identical. And for each file where the R and T versions of the file are different, the target project tree should have three versions of that file each with an altered filename: an R version, a T version and a G version (obtained from the G project tree version in the repo). For example, if a conflicted file has the name foo.java, then there should be in the target project tree these three files: foo_MR.java, foo_MT.java, and foo_MG.java. (You can rename the existing target's foo.java to foo_MT.java, or you can copy it from the T version in the repo.)

The user can manually search for all the conflicted files, via for example the string "_MG", in the target project tree to determine exactly which specific files had a conflict and need special intervention.

Testing

Be sure to provide cases that test for at least one conflict-file resulting in MT, MR, and MG files, and at least one non-conflict file (where the R and T files are identical, or where one of them doesn't exist). Include a test case where there is a common G ancestor at least 2 versions away from the R and the T versions.

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. Normally, this is achieved by refactoring when you are in the local code's area.

Academic Rules

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

Readme File

Same as for project #2.

Submission

Same as for project #2.

Grading

Same as for project #2.