CECS 343 — SWE — VCS Project Part 3

VCS Project Part 2— Merged

Due Friday, May 6, by 11pm.

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. Note that we already have a natural branching effect due to check-out (to a new project tree) coupled with tracking its Mom manifest.

The merge ability lets the user, merge a project tree that is already in the repo (as represented by a manifest file) into a project tree outside the repo.

For example, Fred can merge Jack's changes (checked-in from Jack's project tree) that are in the repo into Fred's current project tree. If the merge succeeds (merge software is only able to handle simple file differences) then Fred can eyeball the merge (maybe also run some tests) and check-in his resulting project tree.

Merge Interface

The user interface for the merge operation is simple. The user needs to indicate the repo manifest file (e.g., by its date-time stamp) of the repo's project tree to be merged, and also indicate the location of the target project tree to merge repo files into.

Merge Conflict

If there is a file version in the target project tree and a different file version in the repo (based on their artifact IDs), then these two file versions are in conflict. Being different, they will need to be merged into a single file that has the differences resolved. The main effort of this project part is to arrange the conflicts so that the user knows what files conflict and where they are located so that he/she can manually merge the conflicted file versions. For extra credit, the VCS can automatically run a merge facility on them (which will do the merge if the conflicts are simple enough), and remove the conflict-files.

Merge Results

After the merge succeeds, the target project tree should have all its conflict-files. Also, a new manifest file should be created which represents the state of the target project tree, with extensions.

The first extension is that new manifest file will have two Moms: the to-be-merged repo manifest file, and the before-merging target's manifest file.

The second extension is that (for the manual 3-way merge) the new manifest file will have three artifact IDs for each file, one for each of the MT, MR, and MG file versions, described below.

Also, a log file should be prepared, named the same as the manifest file with a "_log" appended before the file extension, which lists the pathnames of the conflict-files. (For the extra credit auto-merging, if the merge succeeded then just list the conflict-file without the MT, MR or MG appended.)

Manual 3-Way File Merge

Project part #3 (this current part) includes only setting up for a manual 3-way file merge. This setup consists of finding the three files involved in the merge, copying them to their project tree location, and specially naming them so that the user can easily find them to do a 3-way merge on them. (There is extra credit for automating (actually doing) the 3-way merge for the user; see below.)

Two of the three files for a 3-way merge are easy to find: 1) the file version currently in the target project tree (the "tree-version"), and 2) the corresponding file version in the repo (the "repo-version").

The third file is the most recent file version that is a common ancestor (the "grandpa-file") of those first two file versions. To find the common ancestor file, you will need to trace through the Moms beginning from the target tree manifest ancestry and the Moms of the repo manifest ancestry beginning with the designated repo manifest to find a common ancestor manifest. The grandpa-file version should be in this common ancestor manifest.

To setup for the manual 3-way file merge, you will copy the repo-file and the grandpa-file into the tree-file's location. To make sure that the three files look related (for the user), for the tree-file you will append "_MT" to the tree-file name before the file's extension. Likewise, for the repo-file append "_MR", and for the grandpa-file you will append "_MG". Thus, if the tree-file is named milosh.cpp, the names of the three files will be like this: milosh_MT.cpp, milosh_MR.cpp, and milosh_MG.cpp.

Note that milosh_MT.cpp is the tree-file and will already be in the target project tree, as milosh.cpp; it just needs to be renamed. The milosh_MR.cpp file will be in the requested repo's manifest (under the milosh.cpp/ leaf folder) named by its artifact ID; and needs to be copied to the target project tree and renamed. The milosh_MG.cpp file will also be in the repo, but in the common manifest (also under the milosh.cpp/ leaf folder) named by its artifact ID; and needs to be copied to the target project tree and renamed.

When the dust settles, the target project tree will have all three files (the MT, MR, and MG files) together. The fact that they have these extra name adjustments will clue the user as to which files need to be merged. Presumably, the user will run a 3-way merge program to produce the correct merged milosh.cpp file from these three (and then discard these three input files).

Automated 3-Way File Merge

For extra credit, after the files are copied to the project tree with their MT, MR, and MG names, you will automatically run a 3-way merge facility on them to produce the correct merged file, and discard the three input files. Do this for each file version conflict between the target project tree and the requested repo manifest project tree.

Your team can either incorporate an existing file-merge facility, or you can build from scratch a very simple line-by-line function.

Coding Standards

This is a class in S/W Engineering. In spite of that, we emphasize working S/W. However, getting to working S/W fast often leaves technical debt. Technical debt will rapidly turn into Bad Code if left too long to fester. Therefore, as this is the second rapid delivery, the technical debt must be paid, approximately in full: the source code should comply with the published class S/W Engineering coding standards. Especially, you should be reasonably close with the Active Line counts (which can be confusing for beginners) in your function header comments.

Testing

Be sure to provide cases that test for at least one conflict-file, at least one non-conflict file where the MT and MR files have the same artifact ID, and at least one non-conflict file each where the other file (MT or MR) doesn't exist.

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.

CECS 343 — SWE — VCS Project Part 3

Readme File

You should provide a README.txt text file that includes the class and section, your (team) name, the project/program name, instructions for building, instructions for use, any extra features, and any known bugs to avoid. Be clear in your instruction on how to build and use the project by providing instructions a novice programmer would understand. If there are any external dependencies for building, the README must also list them and how to find and incorporate them. Usage should include an example invocation.

A README would cover the following:

- •Program name
- •Your Name (authors, team)
- •Contact info (email)
- •Class number, Section (eg, 01),
- •"Project Part" and its number
- •Intro (see the Introduction section, above)
- •External Requirements
- •Build, Installation, and Setup
- •Usage
- •Extra Features
- •Bugs

Academic Rules

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

Submission

Put the README, sample, and your source files (and any other text or pdf files) in a folder. Name the folder with the class number and project number (eg, "`p3`") as a prefix (eg "`343-p3-`") and with a suffix of your team name. (Include no executable or "object" files.) Then zip up this folder. Name the .zip file the same as the folder. Thus, if your team name is RAX and this is for project #3 your folder and zip file would have this name: "`343-p3-RAX`".

Team Individual Evaluation

As discussed in class, team is only a team if everyone "pulls their weight" and treats each other somewhat fairly. If one or more members of your team does not seem to be "pulling" his/her weight on the project, please email me separately from the project submission and briefly describe the situation.

Grading

- 70% for compiling (if required) and executing correctly with no errors or warnings
- 30% for Coding Stds level: clean and well-documented code