362 — SWE Fundamentals — VCS Project 2

VCS Project 2— Check-Out & Check-In

Introduction

This is the second part of our VCS (Version Control System) project. In this project part, in C++, we add three new features: **check-out**, **check-in** (mostly already done), and **labeling**.

Label Command

The **labeling** feature allows the user to add a **label** (a text string) to a manifest file, in order to make it easier to remember and use when issuing commands to our VCS. A manifest file must support up to four (4) different **labels** at the same time (more is okay). We can presume that the user is nice and always supplies a unique **label** – so we don't have to check for the **label** already existing in some other manifest file.

A **label** is supposed to uniquely ID a manifest. We can also assume that only the first 30 characters of the **label** (a string) are needed. To add a **label** to a manifest file, the user uses a **label command**, and along with a **label** string argument he/she must also specify either the target manifest's filename or an existing **label** within that manifest. Once a manifest is **labeled**, the user can refer to the manifest by that name in any other VCS commands that require specifying a manifest file. The user never has to add a **label**, or to use a **label**; they can always refer to the manifest file the "long way".

Check-Out Command

The **check-out** ability lets a user recreate a specific version of the project tree. They do this by selecting a particular manifest file in the repo as an argument. Of course, each manifest file specifies every version of every file from a particular version of a project tree.

Note that a given repo folder (the entire repository) only deals with one project (e.g., the Skyrocket project, or the Red-Bunny project, or the Halo project), but can contain many versions of that one project. (If two different projects need VCS, the must each be given their own separate repo folders.)

On **check-out**, the specified project version is recreated by installing its files in a tree in an empty target folder, which the user also provides as an argument. We can assume that the folder is empty. The **check-out** command also creates a new manifest file of the checked out version (for the repo's records) in the repo. The user should be able to specify the manifest file using a **label**, if it has one.

Check-In Command

The **check-in** ability lets the user update the repository (repo) with a different version of the project; meaning you have to add into the repo any changed files in the project tree. So, each **check-in** is a (potentially) different "version" of the project tree, and you create a new manifest file for this different version. The set of the repo's manifest files allow the user to track the modification history for a given project tree, through various project versions, all the way back to the repo's creation. Note that we assume **labels** are forever (the user doesn't remove a **label**).

The **folder** containing a version of the project that the user specifies as an argument to the **check-in** command must have **earlier** been the **target** of a **check-out** command, and we will assume that this is always true. Therefore, in the repo's manifest files, we can trace from a given **check-in** from a folder back to the original **check-out** into that same folder. Note that your manifest files should reflect this ability, as it will be needed later.

362 — SWE Fundamentals — VCS Project 2

Extra Tracking

As an implementation technique, it is reasonable (but not required) to put a dot-file (e.t., ".444-vcs") in the user's project tree top folder to indicate the latest repo manifest associated with this folder; and assume that the user does not alter this dot-file. Dot-files are a traditional Unix technique to provide local control information to a program. Note that such a dot-file is not part of the user's project tree, and thus it should not be copied to/from the repo.

Team

The team may contain up to 3 members, and may be different from the project #1 team. For a new team, pick an appropriate three-letter name in the same ways as was done in project #1.

User Arguments

For both **check-in** and **check-out**, the user will supply a source folder and a target folder (the appropriate one of which will be the repo folder, of course).

On **check-out** the source folder is the repo folder, and the target folder is an empty folder to receive a fresh copy of the specified version of the project tree. Note that the target folder should have the same name as the original project folder name, but will have a different path to that folder name. In addition, on **check-out**, the user will select the manifest of the desired project tree (that was earlier checked in to the repo, or was the original creation manifest).

On **check-in**, the source folder is an existing project tree top folder (with the same name as the project folder used to create the repo, but with a different path), and the target folder is the project's repo folder. (The user might be running several different projects, each with its own repo folder.)

On **labeling**, the user will select the manifest to **label**. We can assume the new **label** is unique and that the manifest has "room" for the **label**.

Check-In Notes

Note that almost all of the **check-in** code has already been developed for the previous Create Repo project part. A few new issues must be handled:

1. For a newer version of a file (with the same project folder relative path) as a file in the repo, the leaf folder will already exist with an earlier version of that file having a different artifact ID. You merely have to add the newer version file into the same leaf folder with its own artifact ID name. You should not remove any existing files in that folder.
2. If a project file has the same computed artifact ID (and project folder relative path) as a file in the repo, then we can presume that it is the same identical file in both placed. So, you need not use this project file to overwrite the existing identical copy in the repo; but you can do such an overwrite if that seems easier to implement.
3. Also, you will create a "**check-in**" manifest file for this command. It will include the command and its arguments as well as the usual manifest information (same as for a "create-repo" command.) Note, if your project #1 manifest didn't include the "create-repo" command and arguments that was used to create it, please upgrade so that that manifest includes these.
4. Regardless of whether a project file has been changed (ie, new artifact ID) or not (ie, duplicate old artifact ID), the file-name and its artifact ID must be recorded in a new manifest file for this **check-in** (with the **check-in** command line arguments and the date-time stamp, of course).

Check-Out Notes

For **check-out**, the user supplies the source repo folder name and an empty target folder name and provides a manifest (representing the specific version of the project files desired). The selection can be either a **label** or the manifest filename. New issues must be handled:

1. You will create a new project tree inside the empty target folder. The files copied from the repo must be those mentioned in the selected manifest.
2. Each needed repo file has an artifact ID as its filename and sits in a sub-folder that is named with its project's filename. This repo artifact file will get copied into the empty target folder's new project tree in the correct path position and must be named with its project filename. For example, we should be able to recreate a project tree if we executed the command sequence:
   a. Create-repo
   b. Accidentally destroy/remove our project tree
   c. **Check-out** to old now-empty project-tree folder by selecting the repo's creation manifest
3. Also, you will create a "**check-out**" manifest file for this command. It will include the **check-out** command and its arguments as well as the date and time and a line for each file checked out (just like the create-repo manifest).

Testing

Test that the code to implement the **Check-in** and **Check-out** works, as follows:

1.On the prior minimal ptree and file:

```
mypt/
  hx.txt  // Contains the letter "H"; hence, you know the checksum.
```

2.On the prior tiny ptree with files:

```
mypt2/
  hx.txt  // As above.
  Stuff/  // A sub-folder
    hello.txt // Contains one line: "Hello world".
    goodbye.txt // Contains two lines: "Good" and then "bye".
```

3.On a small ptree with a sub-folder (called "src") and files:

```
mypt3/
  hx.txt  // As above.
  src/  // A sub-folder.
    main.fool // Main pgm file, Foo Language.
      // Contains one lines:  "Defoo main, darn sock."
    darn.fool // Weird hole-filler component file.
      // Contains two lines:
      // 1. "Defoo darn stuff: set thread.color to param.color;"
      // 2. " find hole; knit hole closed."
```

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. Also, include 2 extra test runs for a (small) "project tree" of your own choice

Also, check that you can add the **labels** "Alice 1" and "Bob #2" to a manifest and then that you can select that manifest for **check-out** with either **label** as well as by specifying its filename.

Readme File, Academic Rules, Submission, Grading: Same as project #1.