## University of Toronto

## Electrical and Computer Engineering

## 

## CSC444 – Software Engineering I

## 

## Assignment 2

## Date: December 2, 2013

## 

## 

## 

|  |  |
| --- | --- |
| **Silverio Miranda** | **Student #: 997472569** |
| **Craig Henriques** | **Student #: 997397302** |
| **Edmund Phung** | **Student #: 997397080** |

## 

## Specification

### 1. Purpose

The purpose of this program is to enable a user to keep track of various versions of source code files throughout their development cycle. In addition to simple versioning, this program also supports features such as branches, which allows for simultaneous feature development and defect correction, as well as merge suggestions for when a user wants to apply changes from one branch to another. Additional features include comments for different version numbers and the ability to list all of the previous versions of a file with their respective comments.

### 2. System Setup

The program was designed, developed and tested for use in a Linux environment. Additionally, the source code control is to be used only with ASCII text files.

All that is required of the user is to copy the program file, named scc.py to the directory which will contain the source code files. The program will set up any other files that it may need in order to handle user requests.

Once a user wishes to begin using the source code control, it can be run with the following command:

|  |
| --- |
| ./scc.py *command* -[flag] [parameters] |

The possible commands and parameters are described below in detail.

### 3. Commands

As input the program takes a specific set of commands and parameters. The following table describes the various commands available to the user, the required and optional parameters, as well as a short description of each.

|  |  |  |
| --- | --- | --- |
| **Command** | **Parameters** | **Description** |
| scc checkin | -f *filename* -c *“comment”* | Checks in a file to the repository with the specified *filename* and *comment*. Note that the comment must be surrounded by quotation marks. The version number is automatically incremented by one. Check-ins done to files in separate branches are not visible to each other. |
| scc checkout | -f *filename* -v *version\_num* | Checks out a file from the repository with the specified *filename* and *version number*. If version number is not specified, then the latest version is checked out. This command is run on the branch that the user is currently editing. |
| scc add | -f *filename* | Adds the specified file with *filename* to the repository and initializes the version number to one. This command is necessary before any other in order for the program to start tracking the changes to the file. |
| scc list | -f *filename* | Prints out all of the versions and comments of the specified file with *filename* since it was either added to the repository or branched from another file. Note that *list* will list the history for whichever branch the user is currently editing. |
| scc branch | -f *filename* -b *branch\_name* | Creates a new branch with *branch\_name* for the specified file with *filename*. This allows for two concurrent versions of the same file to be tracked. The *branch* command will not change the current branch that the user was working in prior to branching. The *switch* command is used for that instead. |
| scc merge | -f *filename* -b *branch\_x* -t *branch\_y* | Creates a suggestion file containing a suggestion on how to merge the most recent checked-in change from *branch\_x* to *branch\_y* for a file with *filename.* It is then the responsibility of the user to verify if the suggested change is correct before checking-in the new file. The suggested file is named *filename. Suggested* and can be fun in the working directory after executing *merge*. |
| scc switch | -f *filename* -b *branch\_name* | Changes to the specified branch with *branch\_name* for the specified file with *filename*. This command may only be used after a *branch* command has been executed. |
| scc list\_branch | -f *filename* | Prints out all of the branches that exist for the specified file with *filename*. This is useful in the event that a user does not recall the branch names they had previously assigned. |

Note that the following flags are also interpreted by the program in place of the above:

|  |  |
| --- | --- |
| **Flag** | **Alternative** |
| -f | --filename |
| -b | --branch |
| -c | --comment |
| -t | --to\_branch |
| -v | --version |

## Design

This document serves to explain and justify the major design decisions that were made in the development of the source code control program. The following two sections describe the overall structure of the program, while the latter sections touch on the key design decisions.

### 1. Directory Structure

The following figure shows the chosen directory structure for the source code control program, including the metadata stored that is invisible to the user.

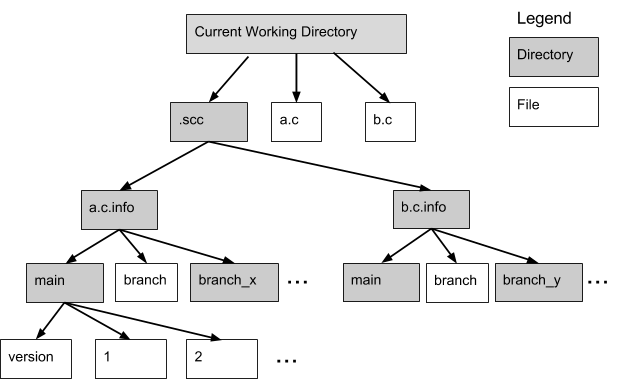


Figure 1. The directory structure of the scc program

The *branch* file is a text file containing the name of the file’s current branch. The *version* file contains the metadata for all versions within the branch and is stored using Python’s Pickle format. It consists of a list (ordered by version) of dictionaries containing the following keys:

|  |  |  |
| --- | --- | --- |
| **Key Name** | **Python Type** | **Description** |
| version | int | Check-in version number (starting at one) |
| comment | string | Check-in comment |
| time | datetime | Check-in time |
| isDiff | boolean | Whether a full copy of the data or only the diffs were saved |

In figure 1, the numbered files contain the text data of each version, where the number refers to the version number. Each of these files may either contain a full copy of the data or may contain only the diffs, relative to the previous version. The first version always contains the full copy. A space-time trade-off exists here in which storing only diffs requires low memory but high computation time while storing only the full versions requires high memory but low computation time. For a good balance between the two, every 5th commit contains the full data while all others contain the diffs. Note that the isDiff key is not completely necessary since we know every 5 commits will contain the full version, but this makes the system more robust to future changes. For example, if in the future it was changed from every 5th commit to every 3rd commit, then existing repositories will still continue to function.

**2.** **Code Structure**

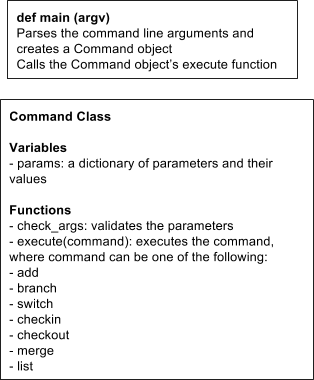


Figure 2. The main code structure for the scc program

The code consists of only one class (the Command class) as that was sufficient for the current complexity of the program. Creating more classes would be over architecting the code and is unnecessary unless the complexity of the program is increased.

The class also contains some private helper methods that are not shown. The commenting style of the code has been formatted to allow pydoc to automatically generate documentation for the code.

### 3. Branch Structure Decision

With respect to branches we decided to use the hierarchy as shown in figure 1 because we felt that it best represented the requirements for this source code control system. Since each file had to have their own branches, we added a level of folders that corresponded to a branch and created a file called ‘branch’ that only contained the string that was the name of the current branch the file was on.

We decided on creating this ‘branch’ file because we needed a fast way to store persistent data. We considered storing the current branch in the branches respective ‘version’ file because this would save space as we would not have an extra file in our directory. However, the downside to this approach would be that we would have to scan through each version file to find out what the current branch was. Since this would scale with the number of branches we decided against it and went with the approach of storing it in a separate file because this would have a constant access time.

Furthermore, whenever we had to update the file on a branch switch we could just overwrite the branch with our updated branch name. In the alternative case we would have to go to the version file of the previous branch and edit and then edit the newer branch file which would take longer.

### 4. Version Metadata File Format Decision

For the ‘version’ file in each branch, Pickle was chosen as the file format over others (such as XML, JSON, or creating our own custom format) due to its ease of use and also because it is a standard file format in Python. It allows objects to be quickly serialized or deserialized without having to worry about the underlying file format.

### 5. Diff and Patch Usage Decision

Our system uses Google’s google-diff-match-patch library for finding diffs and patching. This library was chosen over the UNIX utilities diff and patch as it is multi-platform and allows in-memory diffs and patching, instead of directly operating on files.

### 6. Suggest Merge Algorithm Decision

Our algorithm for suggest merge is as follows given a branch X and Y where we have updated branch X and want to merge the changes into branch Y:

1. Find the difference in between the last two versions of branch X. *This will isolate only the most recent update to branch X*.
2. Take the diff from above and try to apply it to the most recent checkin of branch Y.
3. If the diff was successfully applied then write the file out as a suggested file with filename “*filename.suggested”*. If it was not successfully applied, then warn the user and return.

Note: Threshold values from Google’s google-diff-match-patch were changed to be less strict in what it considered a successful match in order to give the user at least something to work with in tougher merge cases.

We used this algorithm because we felt that it would optimize for the most common case in which merge was called. The case being one in which the user fixed code common to two branches and just wanted to port the changes over. This is the reason that we isolated only the most recent update and tried to apply it.

### 7. Miscellaneous Decisions

The version parameter of the checkout command was chosen to be optional as the user may want to quickly checkout the latest version without having to know the version number. This is useful for situations such as when the user makes some uncommitted changes to files but later decides to discard those changes, which can be done using the checkout command.

## Code

The following is the source code that implements the source code control system.

|  |
| --- |
| **#!/usr/bin/python  #run example : ./scc.py add --filename lol.c  """ This file implements the source code control system as described in the specification and design documents. Documentation for this file can be generated using pydoc. This file can be run using the following command:   ./scc.py command -[flag] [parameters]  """  import os,sys import sys, getopt import pickle import time import datetime import diff\_match\_patch  sys.stderr = sys.stdout  """ Constant defining the number of commits before a full copy of the file is saved (for efficiency) """ full\_save\_frequency = 5  class Command():  """  Class for storing parameters of a command and executing commands  """    def \_\_init\_\_(self,params):  self.params = params  ################################################################### # Helper Functions ###################################################################   def \_\_write\_branch(self,branch):  """ Makes the specified branch the active branch by writing  the desired branch to a file called branch    branch -- the name of the branch to switch to  """    filename = self.params['filename']  my\_path = ".scc/"+filename+".info/"  my\_file = open(my\_path + "branch",'w')  my\_file.write(branch)  my\_file.close()   def \_\_get\_current\_branch(self, filename):  """ Returns the name of the current branch for the given filename    filename -- the file's name  """    fileDirPath = ".scc/"+filename+".info/"   branchFile = open(fileDirPath + "branch",'r')  branchName = branchFile.read()  branchFile.close()   return branchName   def \_\_read\_version\_data(self,filename,branch):  """ Returns the version data consisting of a list of dictionaries  for the given filename and branch    filename -- the file's name  branch -- the branch's name  """    branchPath = ".scc/" + filename + ".info/" + branch + "/"   versionFile = open(branchPath + "version",'r')  data = pickle.load(versionFile)  versionFile.close()   """ Pickle may convert the list to a dict if the list has only  one item, so convert it back to a list if this happens  """  if (type(data) == dict):  data = [data]   return data   def \_\_write\_version\_data(self,filename,branch,data):  """ Writes the version data consisting of a list of dictionaries  for the given filename and branch    filename -- the file's name  branch -- the branch's name  """    branchPath = ".scc/" + filename + ".info/" + branch + "/"   versionFile = open(branchPath + "version", 'w')  pickle.dump(data, versionFile)  versionFile.close()   def \_\_reconstruct\_version\_content(self,filename,branch,requestedVersion):  """ Reconstructs the contents of a file of a given version  by applying the necessary patches    filename -- the file's name  branch -- the branch's name  requestedVersion -- the version to reconstruct  """  versionData = self.\_\_read\_version\_data(filename, branch)   # Find the last version that contained the full content  for versionInfo in reversed(versionData[:requestedVersion]):  if versionInfo["isDiff"] == False:  lastFullVersion = int(versionInfo["version"])  break   branchPath = ".scc/" + filename + ".info/" + branch + "/"   # Check if the version requested already contains the full content  if requestedVersion == lastFullVersion:  # Fetch the full content  fullFile = open(branchPath + str(lastFullVersion),'r')  contents = fullFile.read();  fullFile.close()  else:  # Reconstruct the full content using the diffs  initialFile = open(branchPath + str(lastFullVersion),'r')  contents = initialFile.read();  initialFile.close()   diff = diff\_match\_patch.diff\_match\_patch();   # Apply the patches one by one  for i in range(lastFullVersion + 1, requestedVersion + 1):  patchFile = open(branchPath + str(i),'r')  patch = pickle.load(patchFile)  patchFile.close()   contents = diff.patch\_apply(patch, contents)[0]   return contents   def \_\_create\_file(self,branch):  """ Creates the file in the specified branch and sets its version to 1    branch -- which branch the file should be created in  """    filename = self.params['filename']  my\_path = ".scc/"+filename+".info/"+branch+"/"  os.system("mkdir -p " +my\_path)   versionData = [{ "version": 1, "comment": "First commit", \  "time": datetime.datetime.now(),"isDiff": False }]  self.\_\_write\_version\_data(filename, branch, versionData)   # Since this is the time the file is added , copy the whole file as version 1  os.system("cp "+filename + " "+my\_path+"1")   def \_\_file\_in\_repository(self, filename):  """ Returns whether the file has been added to the repository before  or not    filename -- the file's name  """    my\_path = ".scc/"+filename+".info/"  if(os.path.exists(my\_path)):  return True  else:  return False   def \_\_branch\_exists(self,branch):  """ Returns whether the file has been added/branched before    branch -- Which branch to check for  """    filename = self.params['filename']  my\_path = ".scc/"+filename+".info/"+branch  if(os.path.exists(my\_path)):  return True  else:  return False   def check\_args(self,command):  """ Returns True if the right arguments are supplied for the command  and False if incorrect or extra arguments are supplied    command -- the command name for which to check arguments  """    args = {'add' : ['filename'],  'checkin' : ['filename','comment'],  'checkout' : ['filename','version'],  'checkout' : ['filename'],  'list' : ['filename'],  'list\_branches' : ['filename'],  'branch' : ['filename','branch'],  'merge' : ['filename','branch','to\_branch'],  'switch' : ['filename','branch'],  }  num\_args = 0    for arg in args[command]:  # Make sure the right args are specified  if not arg in self.params:  print "Error: argument "+arg+" required"  sys.exit(1)  num\_args = num\_args + 1   ################################################################### # Primary Functions ###################################################################   def execute(self,command):  """ Runs the command specified    command -- name of version control command to execute  """    #getattr gets a function of this class with name == command  getattr(self, command)()   def add(self):  """ Adds tracking for the current file and sets its version to 1,  will terminate if the file is already under source control  """    #Check if we have info for the file already  if(self.\_\_branch\_exists("main")):  print "Error: file: "+self.params['filename'] +" already added"  sys.exit(1)    self.\_\_create\_file("main")    #Set the branch of the file to main  self.\_\_write\_branch("main")    print "Added file '" + self.params['filename'] + "' at version 1"   def branch(self):  """ Creates a new branch for the specified filename,  will terminate program if the branch already exists  """    branch = self.params['branch'];  filename = self.params['filename'];    if(self.\_\_branch\_exists(branch)):  print "Error: branch '%s' already created" % branch  sys.exit(1)    # Make sure file exists under source control  if not self.\_\_file\_in\_repository(filename):  print "Error: file: '%s' is not in source code control" % filename  return    self.\_\_create\_file(branch)  print "Branch '%s' created" % branch   def switch(self):  """ Switches the branch for the file given,  will terminate if the branch doesn't exist  """    filename = self.params['filename']  branch = self.params['branch']    # Check to see if the branch exists  if not self.\_\_branch\_exists(branch):  print "Error: Can't switch to non-existent branch: '%s'" % branch  sys.exit(1)    # Don't do anything if we are already on the branch  if self.\_\_get\_current\_branch(filename) == branch :  print "Doing nothing, already on branch '%s'" % branch  return    # Update the branch file to switch the branch  self.\_\_write\_branch(self.params['branch'])    # Checkout the most recent version of the file on branch  versionData = self.\_\_read\_version\_data(filename, branch)  version = versionData[-1]["version"]     # Get the content and write it out  content = self.\_\_reconstruct\_version\_content(filename, branch, version)   curFile = open(filename, 'w')  curFile.write(content)  curFile.close()  print "Switched to branch '%s'" % branch   def checkin(self):  """ Checks in the specified file to the repository  """    filename = self.params['filename']   # Make sure file exists under source control  if not self.\_\_file\_in\_repository(filename):  print "Error: file: '"+filename+"' not under source control"  return   branch = self.\_\_get\_current\_branch(filename)  comment = self.params['comment']   # Get the version data for the file  versionData = self.\_\_read\_version\_data(filename, branch)   lastVersion = versionData[-1]["version"]  newVersion = lastVersion + 1   # Compute the diffs between the file in the repository and the \  current file   previousContent = self.\_\_reconstruct\_version\_content(filename, branch, lastVersion)   curFile = open(filename, 'r')  newContent = curFile.read()  curFile.close()   diff = diff\_match\_patch.diff\_match\_patch()  patch = diff.patch\_make(previousContent, newContent)   # Make sure the files are actually different  if not patch:  print "No diffs found, repository already contains latest version"  return   branchPath = ".scc/" + filename + ".info/" + branch + "/"  contentsFile = open(branchPath + str(newVersion), 'w')   # Check whether we should save a full copy or only the diffs  if ((newVersion - 1) % full\_save\_frequency) == 0:  # Save a full copy  contentsFile.write(newContent)   isDiff = False  else:  # Save the diffs  pickle.dump(patch, contentsFile);  isDiff = True   contentsFile.close()   # Create the new version entry  versionEntry = { "version": newVersion, "comment": comment, \  "time": datetime.datetime.now(),"isDiff": isDiff }  versionData.append(versionEntry)   self.\_\_write\_version\_data(filename, branch, versionData)   print "Checked in version " + str(newVersion) + " with comment '" + comment + "'"   def checkout(self):  """ Checks a specific version of a file out of the repository,  if there is no version specified, gets the latest version  """    filename = self.params['filename']   # Make sure file exists under source control  if not self.\_\_file\_in\_repository(filename):  print "Error: file not under source control"  return   branch = self.\_\_get\_current\_branch(filename)   # Get the latest version  versionData = self.\_\_read\_version\_data(filename, branch)  lastVersion = versionData[-1]["version"]   # Check if version was input as parameter  if 'version' in self.params:  version = int(self.params['version'])  # If it wasn't, use latest version  else:  version = lastVersion   # Make sure version is within bounds  if version > lastVersion or version < 1:  print "Error: version number is out of bounds"  return   # Get the content and write it out  content = self.\_\_reconstruct\_version\_content(filename, branch, version)   curFile = open(filename, 'w')  curFile.write(content)  curFile.close()   print "Checked out version " + str(version) + " of file '" + \  filename + "' from branch '" + branch + "'"   def merge(self):  """ Creates a file containing the suggested merge for two different  branches.  Returns an error if no simple merge is possible  """    filename = self.params['filename']  from\_branch = self.params['branch']  to\_branch = self.params['to\_branch']   # Make sure both branches exist  if not self.\_\_branch\_exists(from\_branch):  print "Error: Branch '%s' does not exist" % from\_branch  return  if not self.\_\_branch\_exists(to\_branch):  print "Error: Branch '%s' does not exist" % to\_branch  return   # Get the previous and current version file in the source branch  version\_data = self.\_\_read\_version\_data(filename, from\_branch)  curr\_version = version\_data[-1]["version"]  prev\_version = curr\_version - 1   # Make sure the branch has an update  if prev\_version < 1:  print "Error: Cannot suggest merge on a newly branched file"  return   # Diff the two versions to isolate the last change in a patch  prev\_content = self.\_\_reconstruct\_version\_content(filename, \  from\_branch, prev\_version)  curr\_content = self.\_\_reconstruct\_version\_content(filename, \  from\_branch, curr\_version)   diff = diff\_match\_patch.diff\_match\_patch()  patch = diff.patch\_make(prev\_content, curr\_content)   # Get the contents of the file in the to\_branch  version\_data = self.\_\_read\_version\_data(filename, to\_branch)  curr\_version = version\_data[-1]["version"]  content = self.\_\_reconstruct\_version\_content(filename, to\_branch, curr\_version)   # Patch the file in the latest to branch,and store if it succeeded   content,sucessful= diff.patch\_apply(patch, content)   # Tell the user we couldn't patch if the merge failed  if not sucessful[0]:  print "Error: Unable to merge branches, branch files too different"  return   # Write out the suggested file as filename.suggested  suggested\_file = open(filename + ".suggested",'w')  suggested\_file.write(content)  suggested\_file.close()   print "Merged branch %s to best resemble update in branch %s" \  %( to\_branch, from\_branch)  print "Wrote out suggested merge in %s.suggested" % filename   def list(self):  """ Lists all of the versions (with comments) and time associated   with a file in the current branch  """    filename = self.params['filename']   # Make sure file exists under source control  if not self.\_\_file\_in\_repository(filename):  print "Error: file not under source control"  return   branch = self.\_\_get\_current\_branch(filename)   print "Listing versions for '" + filename + "' in branch '" + branch + "'\n"   # Get the version data and print its contents  data = self.\_\_read\_version\_data(filename, branch)  for version in data:  print "Version " + str(version["version"])  print " Comment: " + version["comment"]  #print "\tDate: " + version["time"].strftime("%c")   def list\_branches(self):  """ Lists all of the different branches that exist for a file  Branch names are the same as the folder names in the directory  containing the metadata for a file  """    filename = self.params['filename']   # Make sure file exists under source control  if not self.\_\_file\_in\_repository(filename):  print "Error: file not under source control"  return   # Set the path to the folder where all the branches are saved  path = ".scc/" + filename + ".info/"   # Print out all of the directory names, i.e. branches  print os.walk(path).next()[1]  def main(argv):  """ Main function of program  Parses the command line arguments and executes the appropriate command  """    # Remove our command out of the arguments and save it  command = argv.pop(0)    # Make sure we have a valid command  if not command in ("add","branch","merge","checkin", \  "checkout","list","switch", "list\_branches"):  sys.exit("Invalid command " + command)  try:  opts, args = getopt.getopt(argv,"hf:c:v:b:t:s:",["filename=",  "comment=","version=","branch=","to\_branch=","--switch"])   except getopt.GetoptError:  print 'Input Error'  sys.exit(1)    # Fill the parameters  params = {}  for opt,arg in opts:  if opt in ("-f","--filename"):  params['filename'] = arg  elif opt in ("-c","--comment"):  params['comment'] = arg  elif opt in ("-v","--version"):  params['version'] = arg  elif opt in ("-b","--branch"):  params['branch'] = arg  elif opt in ("-t","--to\_branch"):  params['to\_branch'] = arg  elif opt in ("-s","--switch"):  params['to\_branch'] = arg   # Call the appropriate command with the arguments  commandObj = Command(params)  commandObj.check\_args(command)  commandObj.execute(command)  if \_\_name\_\_ == "\_\_main\_\_":  main(sys.argv[1:])** |

## Testing Code

The following is the source code that implements the automated testing system used during development and used to output results for the next section describing test results.

|  |
| --- |
| **import os,sys import sys, getopt from subprocess import call  #To run: python test\_scc.py  def validate(test\_num, test):   # Set the paths for the test files  output\_path = "Test\_Actual\_Results/Test%d.txt" %test\_num  expected\_path = "Test\_Expected\_Results/Test%d.txt" %test\_num   # Make a call to the scc program with the desired test conditions  # saving the results to a text file in the output path  os.system(test + " > " + output\_path)   # Open the outputted file and expected results for comparison  expected\_results\_file = open(expected\_path, 'r')  actual\_results\_file = open(output\_path, 'r')   # Read the information contained in the files  expected\_results = expected\_results\_file.readlines()  actual\_results = actual\_results\_file.readlines()   # Print the results based on whether the results match the expected  if (expected\_results == actual\_results):  print "Test %d: Passed" % test\_num  else:  print "Test %d: Failed" % test\_num  print " Command Run: " + test  print " Expected Results: " + str(expected\_results)  print " Actual Results: " + str(actual\_results)   # Close the files and return  expected\_results\_file.close()  actual\_results\_file.close()  def main():   # Set up files for testing in main directory  os.system("cp Test\_Input\_Files/\*.test .")   test\_number = 1   """ Run tests on command parsing """   # Test1: Invalid command  test\_case = "python scc.py fake\_command"  validate (test\_number, test\_case)  test\_number += 1  # Test2: Missing arguments  test\_case = "python scc.py checkin"  validate (test\_number, test\_case)  test\_number += 1   """ Run tests on add """   # Test3: Add a file to be tracked  test\_case = "python scc.py add -f a.test"  validate (test\_number, test\_case)  test\_number += 1   # Test4: Add a file to be tracked that's been added before  test\_case = "python scc.py add -f a.test"  validate (test\_number, test\_case)  test\_number += 1   """ Run tests on branch """   # Test5: Branch a file that's been added  test\_case = "python scc.py branch -f a.test -b test\_branch"  validate (test\_number, test\_case)  test\_number += 1   # Test6: Recreate branch that exists  test\_case = "python scc.py branch -f a.test -b test\_branch"  validate (test\_number, test\_case)  test\_number += 1   # Test7: Branch a file that does not exist  test\_case = "python scc.py branch -f z.test -b test\_branch2"  validate (test\_number, test\_case)  test\_number += 1   """ Run tests on switch """   # Test8: Switch to a branch that exists for a file that exists  test\_case = "python scc.py switch -f a.test -b test\_branch"  validate (test\_number, test\_case)  test\_number += 1   # Test9: Switch to branch that does not exist  test\_case = "python scc.py switch -f a.test -b test\_branch2"  validate (test\_number, test\_case)  test\_number += 1   # Test10: Switch to branch that you are already in  test\_case = "python scc.py switch -f a.test -b test\_branch"  validate (test\_number, test\_case)  test\_number += 1   """ Run tests on checkin """   # Test11: Check in file not under source control  test\_case = "python scc.py checkin -f b.test -c 'file does not exist'"  validate (test\_number, test\_case)  test\_number += 1**  **# Test12: Check in file with no changes, i.e. latest version  test\_case = "python scc.py checkin -f a.test -c 'version 2'"  validate (test\_number, test\_case)  test\_number += 1**  **# Write changes to file a.test before proceeding with next test  os.system("echo 'This is a newline' >> a.test")   # Test13: Check in file with changes (this is on the new\_branch)  test\_case = "python scc.py checkin -f a.test -c 'version 2'"  validate (test\_number, test\_case)  test\_number += 1   """ Run tests on checkout """   # Test14: Check out file that doesn't exist  test\_case = "python scc.py checkout -f z.test"  validate (test\_number, test\_case)  test\_number += 1   # Test15: Check out version that doesn't exist  test\_case = "python scc.py checkout -f a.test -v 3"  validate (test\_number, test\_case)  test\_number += 1   # Test16: Check out file -> version 1  test\_case = "python scc.py checkout -f a.test -v 1"  validate (test\_number, test\_case)  test\_number += 1   # Test17: Check out latest version  test\_case = "python scc.py checkout -f a.test"  validate (test\_number, test\_case)  test\_number += 1   """ Run tests on list """   # Test18: List a file that doesn't exist  test\_case = "python scc.py list -f z.test"  validate (test\_number, test\_case)  test\_number += 1   # Test19: List a file in the current branch (on version 2)  test\_case = "python scc.py list -f a.test"  validate (test\_number, test\_case)  test\_number += 1   # switch to main branch in order to list  os.system("python scc.py switch -f a.test -b main > temp.test")   # Test20: List a file in the main branch (unchanged)  test\_case = "python scc.py list -f a.test"  validate (test\_number, test\_case)  test\_number += 1**  **""" Run tests on list\_branches """   # Test21: List all branches for a file that exists  test\_case = "python scc.py list\_branches -f a.test"  validate (test\_number, test\_case)  test\_number += 1   # Test22: List all branches for a file that doesn't exist  test\_case = "python scc.py list\_branches -f z.test"  validate (test\_number, test\_case)  test\_number += 1   """ Run tests on merge """   # Test23: Merge a branch that doesn't exist  test\_case = "python scc.py merge -f a.test -b fake -t main"  validate (test\_number, test\_case)  test\_number += 1   # Test24: Merge a branch with a change to a different branch  test\_case = "python scc.py merge -f a.test -b test\_branch -t main"  validate (test\_number, test\_case)  test\_number += 1   # Create a new branch for next test  os.system("python scc.py branch -f a.test -b test\_branch2")   # Test25: Merge a branch that was just branched, no changes  test\_case = "python scc.py merge -f a.test -b test\_branch2 -t main"  validate (test\_number, test\_case)  test\_number += 1   # Switch to new branch  os.system("python scc.py switch -f a.test -b test\_branch2")  # Write changes to file a.test before proceeding with next test  os.system("echo 'Small Change!' >> a.test")  # Checkin the file with changes  os.system("python scc.py checkin -f a.test -c 'Rewritten File'")  # Write changes to file a.test before proceeding with next test  os.system("echo 'I rewrote the file!' > a.test")  # Checkin the file with changes  os.system("python scc.py checkin -f a.test -c 'Rewritten File'")  # Switch back to main before merging  os.system("python scc.py switch -f a.test -b main")   # Test26: Try to merge a completely rewritten file  test\_case = "python scc.py merge -f a.test -b test\_branch2 -t main"  validate (test\_number, test\_case)  #test\_number += 1   """ Remove test files from main directory """  os.system("rm \*.test")  os.system("rm -rf .scc")  if \_\_name\_\_ == "\_\_main\_\_":  main()** |

## Testing

This following section describes unit and system tests what were run on the system to verify its functionality.

### 1. UnitTests

The unit tests described below were implemented to test some of the more complex helper functions that were implemented to aid the primary functions such as add,branch,checkin, etc.

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Expected Result** | **Actual Result** | **Comment** |
| **\_reconstruct\_version\_conten**t - *A function that reconstructs a file to a specified version on a specified branch by applying patches*  Tested by storing patches to a file and then seeing if the function could reconstruct them | The version of the file we wished to reconstruct was identical to its original state | The file was reconstructed as expected | This unit test was fundamental because it made sure that we were able to move between versions of files correctly though patches. If this didn’t work errors caused by it would show up in many places |
| **\_read\_version\_data**  *This function parses the version metadata and returns an object containing the information*  Tested by creating a fake(but valid) version file and seeing if it interpreted it correctly | The series of versions specified in the version file got printed out and in the right order | The version and order were maintained as expected | This unit test was necessary since many functions relied on parsing the version file and went through this function to do it |
| **\_\_create\_file**  *This function actually makes our hidden directories and copies the initial version of the file.*  Tested by calling the function on 3 different files | The function would create separate directories for each file and actually copy them in | The directory structure and file contents were present as expected | This was important to test as it verified that our initialization worked. It also confirmed that we were correctly interacting with the OS |

### 

### 2. System Tests

The following tests serve to both test the functionality of each individual function, and also test the system as a whole, as there are dependencies between the commands themselves. These 26 tests cover most of the exit conditions for each of the primary functions. Since each of these tests pass, one can conclude that the code is in good working order with minimal defects.

These tests were all run consecutively, accurately reflecting how the system would be behave under normal use.

All actual results (I/O messages) produced by the program were equal to the expected results (I/O messages). The output of the tester program is included at the end of this document, showing the matching outputted messages for each case. As this is the case, this table is simplified to discuss what is happening at a higher level.

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Expectation &**  **Result ( = Expectation)** | **Comment** |
| **Test1:** Invalid command  *python scc.py fake\_command* | When an invalid command is entered, the program should produce an error and exit.  This is what happens. | This test is essential filter bad input. The program should only accept a specific predefined set of commands in order to maintain expected functionality. |
| **Test2:** Missing arguments  *python scc.py checkin* | When there are missing flags or parameters, the function should not execute, but rather exit with an error.  This is what happens. | Functions expect certain parameters so it is important to verify that the user is inputting all needed information in order to function as expected. |
| **Test 3:** Add a file to be tracked  *python scc.py add -f a.test* | When a file is added a confirmation should be printed and the file should now allow other commands to be run on it. | This is critical to verify since without this command, no other test can be run. |
| **Test 4:** Add a file to be tracked that's been added before  *python scc.py add -f a.test* | Print an error to the user. | Multiple same files cannot be added or there would be version conflicts and unexpected behaviour. |
| **Test 5:** Branch a file that's been added  *python scc.py branch -f a.test -b test\_branch* | Print a confirmation to the user. Any branch operations should now be applicable to the file branched. | This is essential to the accurate results of *switch, merge, list* and above all the user experience to allow for simultaneous feature adding and defect correction. |
| **Test 6:** Recreate branch that exists  *python scc.py branch -f a.test -b test\_branch* | Print an error to the user. | This function would be doing extra work needlessly. Better to avoid this scenario as there could already be files in the branch with potentially resulting undesired behaviour. |
| **Test 7:** Branch a file that does not exist  *python scc.py branch -f z.test -b test\_branch2* | Print an error to the user. | If this is not handled properly, there could be many useless branches created with no content. It also serves as a reminder to the user to add a file first in case they had forgotten, increasing usability. |
| **Test 8:** Switch to a branch that exists for a file that exists  *python scc.py switch -f a.test -b test\_branch* | Print a confirmation message of the switch and the file that was previously branched should now appear in the working directory. | This test is important to ensuring that branches can be used at all. If a user cannot switch branches, it is as if they do not use branches. |
| **Test 9:** Switch to branch that does not exist  *python scc.py switch -f a.test -b test\_branch2* | Print an error to the user. | It is important to catch this issue files could become lost. If there is no branch set up properly then the metadata would also not exist for proper code retrieval. |
| **Test 10:** Switch to branch that you are already in  *python scc.py switch -f a.test -b test\_branch* | Print an error to the user. | This function would be doing extra work needlessly. Better to avoid that to avoid slowdowns for the user with larger systems. |
| **Test 11:** Check in file not under source control  *python scc.py checkin -f b.test -c 'file does not exist'* | Print an error to the user. | This would serve as a reminder message to the user to add the file to source control. This adds to the user experience. |
| **Test 12:** Check in file with no changes, i.e. latest version  *python scc.py checkin -f a.test -c 'version 2'* | Print an error to the user. | This function would be doing extra work needlessly. Better to avoid that to avoid slowdowns for the user with larger files. |
| **Test 13:** Check in file with changes (this is on the new\_branch)  *python scc.py checkin -f a.test -c 'version 2'* | One a file has been checked in, it should be given a new version number, appear in list, and allow other users to check it out. A confirmation message should also be printed out for the user. | This command is one of two that form the basis for version control (checkin and checkout). This command must function or else a user has no way of retrieving previously saved code version. It must be checked to ensure proper version numbers are assigned. |
| **Test 14:**  Check out file that doesn't exist  *python scc.py checkout -f z.test* | Print an error to the user. | This is important to check as there would be garbage information printed to the user, reducing their user experience. |
| **Test 15:**  Check out version that doesn't exist  *python scc.py checkout -f a.test -v 3* | Print an error to the user. | This is important to check as there would be garbage information printed to the user, reducing their user experience. |
| **Test 16:** Check out file -> version 1, (i.e. reverting changes)  *python scc.py checkout -f a.test -v 1* | Once an older file is checked out, it should be visible in the working directory of the user, whether it was newly added or reverting an older version. A confirmation should also be printed out for the user. | This command is one of two that form the basis for version control (checkin and checkout). If the user has no way of retrieving a file that was previously checked in, the system is completely useless to them. |
| **Test 17:** Check out latest version  *python scc.py checkout -f a.test* | Same as above. | Same as above. |
| **Test 18:** List all versions of a file that doesn't exist  *python scc.py list -f z.test* | Print an error to the user. | This is important to check as there would be garbage information printed to the user, reducing their user experience. |
| **Test 19:**  List all versions of a file in the current branch  *python scc.py list -f a.test* | Should print all versions plus comments for the file in the current branch. In this case there will be two versions. | This test is necessary to ensure that the user can remember which changes he made over time. It also allows them to quickly decide which version of a file they would need to check-out instead of trial and error. |
| **Test 20:** List all versions of a file in the main branch (unchanged)  *python scc.py list -f a.test* | Same as above. In this case there will only be 1 version. | Same as above. |
| **Test 21:** List all branches for a file that exists  *python scc.py list\_branches -f a.test* | Should list all of the branch IDs that were ever created for the specified file. In this case it outputs two names. | This test is important for user convenience. If a user forgets a name of a branch, he has no other way of ever retrieving it, possibly losing significant changes that were made. |
| **Test 22:** List all branches for a file that doesn't exist  *python scc.py list\_branches -f z.test* | Print an error to the user. | This is important to check as there would be garbage information printed to the user, reducing their user experience. |
| **Test 23:** Merge a branch that doesn't exist  *python scc.py merge -f a.test -b fake -t main* | Print an error to the user. | This is important to check as there would be garbage information printed to the user, reducing their user experience. |
| **Test 24:** Merge a branch with a change to a different branch  *python scc.py merge -f a.test -b test\_branch -t main* | Should output a .suggestion file with suggestions based on the last change to branch X that will be applied to branch Y. Indeed the suggestion is created:  a.test (main):  “hello world”  a.test (test\_branch):  “hello world  This is a newline”  a.test.suggested:  “hello world  This is a newline”  Should also print a confirmation to the user. | Merge is an essential function, and as such it requires a lot of testing to ensure that the suggestions made are accurate more of the time, or users will be thoroughly frustrated with having to apply changes manually each time. |
| **Test 25:** Merge a branch that was just branched, no changes  *python scc.py merge -f a.test -b test\_branch2 -t main* | Print an error to the user. | This function would be doing extra work needlessly. Better to avoid that to avoid slowdowns for the user with larger files. |
| **Test 26:** Try to merge a completely rewritten file  *python scc.py merge -f a.test -b test\_branch2 -t main* | Same as Test 24, with different input branches:  a.test (main):  “hello world”  a.test (test\_branch2):  “I rewrote the file!”  a.test.suggested:  “I rewrote the file!” | Same as Test 24. |

**3.** **Tester Program Output**

The following is the output of the tester program, showing that expect output and actual output are equal for each test case mentioned above.

|  |
| --- |
| Test 1: Passed  Command Run: python scc.py fake\_command  Expected Results: ['Invalid command fake\_command\n']  Actual Results: ['Invalid command fake\_command\n'] Test 2: Passed  Command Run: python scc.py checkin  Expected Results: ['Error: argument filename required\n']  Actual Results: ['Error: argument filename required\n'] Test 3: Passed  Command Run: python scc.py add -f a.test  Expected Results: ["Added file 'a.test' at version 1\n"]  Actual Results: ["Added file 'a.test' at version 1\n"] Test 4: Passed  Command Run: python scc.py add -f a.test  Expected Results: ['Error: file: a.test already added\n']  Actual Results: ['Error: file: a.test already added\n'] Test 5: Passed  Command Run: python scc.py branch -f a.test -b test\_branch  Expected Results: ["Branch 'test\_branch' created\n"]  Actual Results: ["Branch 'test\_branch' created\n"] Test 6: Passed  Command Run: python scc.py branch -f a.test -b test\_branch  Expected Results: ["Error: branch 'test\_branch' already created\n"]  Actual Results: ["Error: branch 'test\_branch' already created\n"] Test 7: Passed  Command Run: python scc.py branch -f z.test -b test\_branch2  Expected Results: ["Error: file: 'z.test' is not in source code control\n"]  Actual Results: ["Error: file: 'z.test' is not in source code control\n"] Test 8: Passed  Command Run: python scc.py switch -f a.test -b test\_branch  Expected Results: ["Switched to branch 'test\_branch'\n"]  Actual Results: ["Switched to branch 'test\_branch'\n"] Test 9: Passed  Command Run: python scc.py switch -f a.test -b test\_branch2  Expected Results: ["Error: Can't switch to non-existant branch: 'test\_branch2'\n"]  Actual Results: ["Error: Can't switch to non-existant branch: 'test\_branch2'\n"] Test 10: Passed  Command Run: python scc.py switch -f a.test -b test\_branch  Expected Results: ["Doing nothing, already on branch 'test\_branch'\n"]  Actual Results: ["Doing nothing, already on branch 'test\_branch'\n"] Test 11: Passed  Command Run: python scc.py checkin -f b.test -c 'file does not exist'  Expected Results: ["Error: file: 'b.test' not under source control\n"]  Actual Results: ["Error: file: 'b.test' not under source control\n"] Test 12: Passed  Command Run: python scc.py checkin -f a.test -c 'version 2'  Expected Results: ['No diffs found, repository already contains latest version\n']  Actual Results: ['No diffs found, repository already contains latest version\n'] Test 13: Passed  Command Run: python scc.py checkin -f a.test -c 'version 2'  Expected Results: ["Checked in version 2 with comment 'version 2'\n"]  Actual Results: ["Checked in version 2 with comment 'version 2'\n"] Test 14: Passed  Command Run: python scc.py checkout -f z.test  Expected Results: ['Error: file not under source control\n']  Actual Results: ['Error: file not under source control\n'] Test 15: Passed  Command Run: python scc.py checkout -f a.test -v 3  Expected Results: ['Error: version number is out of bounds\n']  Actual Results: ['Error: version number is out of bounds\n'] Test 16: Passed  Command Run: python scc.py checkout -f a.test -v 1  Expected Results: ["Checked out version 1 of file 'a.test' from branch 'test\_branch'\n"]  Actual Results: ["Checked out version 1 of file 'a.test' from branch 'test\_branch'\n"] Test 17: Passed  Command Run: python scc.py checkout -f a.test  Expected Results: ["Checked out version 2 of file 'a.test' from branch 'test\_branch'\n"]  Actual Results: ["Checked out version 2 of file 'a.test' from branch 'test\_branch'\n"] Test 18: Passed  Command Run: python scc.py list -f z.test  Expected Results: ['Error: file not under source control\n']  Actual Results: ['Error: file not under source control\n'] Test 19: Passed  Command Run: python scc.py list -f a.test  Expected Results: ["Listing versions for 'a.test' in branch 'test\_branch'\n", '\n', 'Version 1\n', ' Comment: First commit\n', 'Version 2\n', ' Comment: version 2\n']  Actual Results: ["Listing versions for 'a.test' in branch 'test\_branch'\n", '\n', 'Version 1\n', ' Comment: First commit\n', 'Version 2\n', ' Comment: version 2\n'] Test 20: Passed  Command Run: python scc.py list -f a.test  Expected Results: ["Listing versions for 'a.test' in branch 'main'\n", '\n', 'Version 1\n', ' Comment: First commit\n']  Actual Results: ["Listing versions for 'a.test' in branch 'main'\n", '\n', 'Version 1\n', ' Comment: First commit\n'] Test 21: Passed  Command Run: python scc.py list\_branches -f a.test  Expected Results: ["['main', 'test\_branch']\n"]  Actual Results: ["['main', 'test\_branch']\n"] Test 22: Passed  Command Run: python scc.py list\_branches -f z.test  Expected Results: ['Error: file not under source control\n']  Actual Results: ['Error: file not under source control\n'] Test 23: Passed  Command Run: python scc.py merge -f a.test -b fake -t main  Expected Results: ["Error: Branch 'fake' does not exist\n"]  Actual Results: ["Error: Branch 'fake' does not exist\n"] Test 24: Passed  Command Run: python scc.py merge -f a.test -b test\_branch -t main  Expected Results: ['Merged branch main to best resemble update in branch test\_branch\n', 'Wrote out suggested merge in a.test.suggested\n']  Actual Results: ['Merged branch main to best resemble update in branch test\_branch\n', 'Wrote out suggested merge in a.test.suggested\n'] Test 25: Passed  Command Run: python scc.py merge -f a.test -b test\_branch2 -t main  Expected Results: ['Error: Cannot suggest merge on a newly branched file\n']  Actual Results: ['Error: Cannot suggest merge on a newly branched file\n'] Test 26: Passed  Command Run: python scc.py merge -f a.test -b test\_branch2 -t main  Expected Results: ['Merged branch main to best resemble update in branch test\_branch2\n', 'Wrote out suggested merge in a.test.suggested\n']  Actual Results: ['Merged branch main to best resemble update in branch test\_branch2\n', 'Wrote out suggested merge in a.test.suggested\n'] |