Git Client Tool Validation

Department Name

March 4, 2022

Abstract

Throughout this document Git is the Configuration Item that is being validated and Report Package is Git Report Package. Git is considered validated after this Report Package has been executed, test evidence has been obtained, and that test evidence supports the conclusion the Intended Use Requirements (IUR) have been satisfied.

Contents

1 Introduction

1.1 Overview

Git is a distributed revision control system with an emphasis on speed, data integrity, and support for distributed, non-linear workflows. Git allows repository clones to act as fully functioning repositories complete with history and full revision tracking capabilities that are not dependent on network access or a central server.

1.2 Purpose

This Report Package is a detailed record that provides a Configuration Item overview, a list of its Intended Use Requirements (IUR), one or more Test Reports, evidence the Test Reports ran, along with the output produced by the Test Report. The Test Report includes a pass/fail result for each Test Step and Test Report, a statement indicating the Configuration Item has a Configuration Identification and a conclusion that the Configuration Item has been validated for its intended use.

1.3 Scope

This Report Package applies to Company medical device software projects that have determined a Configuration Item must be validated for its intended use. This Report Package covers activities associated with validating a Configuration Item for its intended use requirements.

1.4 Deviations

The process governing the creation of this protocol and report deviates from the normal standard operating procedure (SOP005 Validation of Computerized Systems). This document combines both the protocol and the report. Normally the protocol is released first and report is released after the protocol is executed. This document represents an automated protocol execution facilitated through the use of automation scripting and software. The review of a paper protocol and pre-approval of said protocol does not satisfy the need to review the automated components used for the generation of this document. As a result, the automated components which codify the actual test protocol are reviewed by a technical approver as this document and the components are developed. This technical approver is an approver of this document and their approval indicates the automated components effectively test the article under test to meet the intended use as specified in the user requirements.

Additionally data obtained from the execution of the protocol is collected and presented in the grey boxes as objective evidence from the automated test application. Normally this would not be presented together with the protocol, but given this is an automated process in a combined document; this is an effective means of retaining and presenting the objective evidence for review and approval.

Finally, presenting the protocol and the report together allows for a single step automation process that can be easily maintained and re-executed. Re-execution is often desired due to changes to the article under test or changes to user needs.

1.5 Tool Validation Objectives

Document 123-VNV-056022 Validation Determination report provides a Determination of Validation decision tree and Determination of Level of Risk and Validation Rigor decision tree to aid a Development Team when assessing the need for validation. 123-VNV-056022 was updated to indicate this tool required Validation and the Level of Risk needed. The steps below describe the steps used to validate a tool.

1. Describe the intended use of the tool.
2. Set the purpose and scope for the tool validation effort.
3. Enumerate intended use requirements.
4. Disclose compliance criteria.
5. Define Tool validation acceptance criteria.

6. Identify responsible persons and their roles.
7. Document required deliverables.
8. Define specific test steps and test steps to confirm that the Tool's intended use requirements have been met.
9. Collect test evidence.
10. Record Tool validation conclusion.

1.6 General Terms

Configuration Control The systematic process for managing changes to and established baseline.

Configuration Identification A unique identifier used to associate a collection of software artifacts.

Configuration Items Software source code, executables, build scripts, and other software development and software test artifacts relevant to creating and maintaining a software project.

Configuration Status Accounting The recording and reporting of the information needed to effectively manage the software and documentation components of a software project.

Report Package A detailed record that provides a Configuration Item overview, a list of its Intended Use Requirements (IUR), one or more Test Reports, evidence the Test Reports ran along with the output produced by Test Report including a pass/fail result for each Test Step and Test Report, and a statement indicating the Configuration Item has a Configuration Identification, and conclusion that the Configuration Item has been validated for its intended use.

Test Plan A test plan is a collection of one or more test suites a tester has determined to use to challenge requirements.

Test Suite A test suite is a collection of one or more test cases a tester has determined to use to challenge requirements.

Test Case A test case is a set of conditions under which a tester will determine whether the test is working as it was originally established for it to do.

Test Step A unique test identifier with predetermined expectation, confirmation criteria, and pass/fail result.

Test Report A test report consists of Detailed instructions for the set-up, execution, and evaluation of results for a given test. The test protocol may include one or more test cases for which the steps of the protocol will repeat with different input data. Test cases are chosen to ensure that corner cases in the code and data structures are covered. A test protocol may be a script that is automatically run by the computer.

1.7 Configuration Item Specific Terms

Branch A line of development.

Checkout Updates files in the working tree to match the version in a remote repository.

Clone To make a local copy of a remote repository.

Commit Record changes to a repository.

HEAD A pointer to the commit being worked on.

log A record of commit entries.

merge Join two or more development histories together. The index or the specified tree. If no paths are given, git checkout will also update the HEAD to set the specified branch as the current branch.

tag - annotated Tag objects are called 'annotated' tags; they contain a creation date, the tagger name and email, a tagging message, and an optional GnuPG signature.

tag - lightweight Simply a name for a commit object. The object is usually a commit object.

1.8 General Acronyms

FDA Food and Drug Administration

IUR Intended Use Requirements

LMS Learning Management System

SOP Standard Operating Procedure

SOUP Software Of Unknown Provenance

1.9 Configuration Item Specific Acronyms

n/a Not applicable

1.10 References

- SOP004 Software Devopment Procedure
- SOP003 Software Configuration Management
- SOP001 Good Documentation Practices
- SOP005 Validation of Computerized Systems
- SOP002 Change Management

1.11 Training

Company's training records are stored in the Quality Management System. Additional training is not required because this is an automated test that is executed by the automated testing platform. SOP004 Software Devopment Procedure provides training required to create, maintain, and execute this testing protocol.

1.12 Tool Validation Test Approach

This Test Plan describes a series of Test Suites, Test Cases, and Test Steps. When executed, each Test Step determines if the Configuration Item satisfies one or more system requirements. When a Test Step indicates that the system requirements are satisfied, the Test Step's result is "pass". Otherwise, the Test Step's result is "fail". The computer records all "pass" and "fail" results in the Test Plan record. The Configuration Item is considered verified when all Test Steps are executed and the Test Plan record contains no "fail" results. Each Test Step that results in a deviation, observation, incident, or failure shall be represented in the final report.

1.13 Configuration Management

When a Configuration Item is changed, we will review the manufacturer's release notes or our design history file (DHF) to determine if regression testing or adjustments to this Report Package is necessary. We will verify the changes do not impact product operation, product quality, or quality decision made prior to performing the upgrade.

1.14 Test Plan Instructions

This Test Plan describes Test Suits, Test Cases, and Test Steps that demonstrate how the Configuration Item satisfies the IUR. Each Test Plan describes any setup criteria needed to conduct the test. Each Test Plan contains a list of IUR's and the steps that demonstrate how the Configuration Item satisfies the IUR. Each Test Step is marked passed or failed as it is completed. Each Test Plan is marked passed when all Test Steps pass or failed if a single Test Step fails. Failures are addressed per SOP002 Change Management. This serves as a record of the completed test.

Test Plans are automatically run by the computer, generating a report in PDF format. This Report Package is reviewed prior to execution per SOP004 Software Devopment Procedure. The Report Package

is routed and archived in the Quality Management System. When it becomes necessary to annotate a computer generated document SOP001 Good Documentation Practices must be followed.

1.15 Test Plan Storage and Review

This Test Plan is part of a Company's automated validation framework. The framework consists of following parts:

LaTeX files are used to provide an Abstract, Introduction, Intended Use Requirements, Test Plan Overview, Test Equipment, Configuration Item Validation, Conclusion, and Change Summary. LaTeX files are converted assembled into PDF documents. PDF documents are routed using the Company's document management system for approval.

Ruby software is used to run the automated framework to collect test evidence.

Git is used as the storage repository for LaTeX & YAML files, a Git pull-request is used to review the LaTeX & YAML files prior to use.

Evidence Test Plan output includes one Test Suite, Test Plan, and Test Step, and Test Evidence.

YAML files define the Test Plan, Test Suite, and Test Steps that are processed to generate test evidence.

2 Intended Use

Intended use requirements are defined using the following story format:
As a <type of user>, I want <some goal> so that <some reason>

IUR-000 As a developer, I want to configure a Git client appropriate to my operating system so that I can interact with Git server.

IUR-001 As a developer, I want to create and modify a local repository so that I can backup my daily work products.

IUR-002 As a developer, I want to add, modify and delete files from a local repository so that I can preserve my daily work products.

IUR-003 As a developer, I want to obtain files from a remote repository so that I can synchronize my work with my team mates.

IUR-004 As a developer, I want to annotate changes made to a local repository so that I track my work products.

IUR-005 As a developer, I want to isolate my work products so that I can work on multiple assignments concurrently.

IUR-006 As a developer, I want know the status of my local repository so that I can determine the state of my work products with respect to revision control.

IUR-007 As a developer, I want to share my work products with other developers so that I can support team collaboration.

IUR-008 As a developer, I want to know the difference between my local repository and a remote repository so that I can correctly integrate my work with the work of others.

IUR-009 As a developer, I want to apply a tag to a git-commit so that I can re-create a branch by tag.

IUR-010 As an operations personnel, I want to use a source control system to manage and track relevant system state and configuration within a device.

3 Test Plan Overview

This section describes Test Plans, Test Suites, Test Cases, and Test Steps that demonstrate how a Configuration Item satisfies the IUR. Each Test Plan describes any setup criteria needed to conduct the Test Steps. Each Test Plan contains a list of IUR's and the Test Steps that demonstrate how the Configuration Item satisfies the IUR. Each Test Step is marked passed or failed as it is completed. Each Test Plan is marked passed when all Test Steps pass or failed if a single Test Step fails. This serves as a record of completed Test Plans and Test Steps.
Each Test Plan is described in its own section. The order the Test Plans are listed is the order they are run. Each Test Plan defines:

name	Each Plan, Suite, and Case has a unique name.
purpose	Each Plan, Suite, and Case has a purpose.
Test Steps	Each step has a confirmation and expectation along with the command needed to challenge the IUR.
Objective Evidence	A record the Test Plan was run along with any evidence collected while the Test Steps were run.
Traceability	Suites and Cases are traced to an IUR that is challenged. IUR can be traced to multiple Suites and Cases.

Each Test Plan, Test Suite, Test Case, and Test Step has been designed to be run by the computer. However, a person may choose to manually run the Test Step, save the test results, and generate this test report as specified in the appropriate design documentation. The example below runs these commands:

1. git help
2. cat .gitconfig

The output from both commands are written to the system console.

```
plan:
  name: A Test Plan Name
  purpose: purpose of the plan

suite:
  name: A Test Suite Name
  purpose: a suite purpose
  requirement: IUR01 and IUR02

  - case:
    name: A Test Case name
    purpose: A Test Case purpose
    steps:
      - confirm: Confirm git help is written to the console output.
        expectation: Git help is displayed.
        command: git
        argument: help

      - confirm: Confirm .git config is written to the console.
        expectation: .gitconfig is written to the console output.
        command: cat
        argument: .gitconfig
```

4 Prerequisites

1. A valid Git username and password. Request these from your Git administrator.
2. Read and write access to https://bitbucket-vial.intra.fresenius.com/projects/SAN/repos/quicksort
3. A private / public SSH key.
4. Your public key needs to be upload to your Git account. Request assistance from your Git administrator.
5. Administrative rights to your computer. Contact your computer systems administrator.
6. Your computer requires HTTP, HTTPS, and SSH access to the Internet. Contact your network administrator.
7. Configure your /.ssh/config file to reference Git your Git Server.

```
# Access to HostName when disconnected from a Company network.
Host github.com
    User username
    IdentityFile ~/.ssh/github_rsa

# All hosts
Host *
    ServerAliveInterval 300
    IdentityFile ~/.ssh/github_rsa
    User gahoward
    Port 22
```

5 Test Evidence

The Company's automation framework assembles the content in this section. The section has one or more Test Plans, Test Suites, Test Cases, and Test Evidence. The evidence provided is used to conclude the Tool has met the Intended Use Requirements.

5.1 Test Plan: Git Client Validation Plan

Purpose: This plan uses white box methods. The output is sufficient for DHF purposes. However, the output is generally only used by engineers.

5.1.1 Test Suite: Git Client Setup

Purpose: This test suite ensures the Git client has been installed and is operational.

Requirement: IUR-000, IUR-001, IUR-002, IUR-003, IUR-004, IUR-005, and IUR-006.

5.1.2 Test Case: Check Install

Purpose: This protocol is used to demonstrate that the Git Client program has been properly installed and configured. This is accomplished by asking the operating system where the program is installed.

Requirement: IUR-000

Step: 1

Confirm: The git client has been installed.

Expectation: git installation location is listed.

Command: echo which git

Test Result: PASS

Evidence: Starts on next line.

```
1 which git
```

Step: 2

Confirm: The git client version is displayed.

Expectation: git reports it version on the console.

Command: echo git –version

Test Result: PASS

Evidence: Starts on next line.

```
1 git ––version
```

5.1.3 Test Case: Configure Git

Purpose: This protocol is used to demonstrate that the Git software has been properly installed and configured. This is done by using Git to configure the .gitconfig file and then printing the results of the changes made to .gitconfig.

Requirement: IUR-000

Step: 1

Confirm: Confirm a developer is able to access Git Help.

Expectation: Git help is displayed.

Command: echo git help

Test Result: PASS

Evidence: Starts on next line.

```
1 git help
```

Step: 2

Confirm: A developer is able to initialize the repository.

Expectation: No error.

Command: echo git init

Test Result: PASS

Evidence: Starts on next line.

```
1 git init
```

Step: 3

Confirm: A developer is able to configure a Git code pager.

Expectation: No error.

Command: echo git config –local code.pager cat

Test Result: PASS

Evidence: n/a

```
git config —local code.pager cat
```

Step: 4

Confirm: A developer is able to configure a Git user name.

Expectation: No error.

Command: echo git config –local user.name "Gary A. Howard"

Test Result: PASS

Evidence: n/a

```
git config —local user.name Gary A. Howard
```

Step: 5

Confirm: A developer is able to configure a Git user email address.

Expectation: No error.

Command: echo git config –local user.email gary.a.howard@mac.com

Test Result: PASS

Evidence: n/a

```
1 git config —local user.email gary.a.howard@mac.com
```

Step: 6

Confirm: A developer is able to configure a Git credentials.

Expectation: No error.

Command: echo git config –local credential.helper store

Test Result: PASS

Evidence: n/a

```
git config —local credential.helper store
```

Step: 7

Confirm: A developer is able to disable color output to the terminal.

Expectation: No error.

Command: echo git config –local color.ui false

Test Result: PASS

Evidence: n/a

```
git config —local color.ui false
```

Step: 8

Confirm: Confirm a developer has a properly configured .gitconfig file.

Expectation: .gitconfig file is displayed.

Command: echo cat .git/config

Test Result: PASS

Evidence: Starts on next line.

```
1 cat .git/config
```

5.1.4 Test Case: Create a Local Git Repository

Purpose: This test case is used to demonstrate a local Git repository is properly initialized, files are added to a local working copy of the repository causing them to become tracked by Git, tracked files are committed to the local repository, and files not tracked by Git are removed from the Git repository working area.

Requirement: IUR-001, IUR-002, IUR-004, IUR-005, and IUR-006

Step: 1

Confirm: Confirm a developer is able to touch a file named f1.

Expectation: No error.

Command: echo touch f1

Test Result: PASS

Evidence: n/a

```
1 touch f1
```

Step: 2

Confirm: Confirm a developer is able to touch a file named f2.

Expectation: No error.

Command: echo touch f2

Test Result: PASS

Evidence: n/a

```
1 touch f2
```

Step: 3

Confirm: Confirm a developer is able to list newly touched files.

Expectation: Directory listing has newly touched files.

Command: echo ls -la

Test Result: PASS

Evidence: Starts on next line.

```
1 ls -la
```

Step: 4

Confirm: Confirm a developer is able to initialize a local repo.

Expectation: No error.

Command: echo git init

Test Result: PASS

Evidence: Starts on next line.

```
1 git init
```

Step: 5

Confirm: A developer is able to disable color output to the terminal.

Expectation: No error.

Command: echo git config –local color.ui false

Test Result: PASS

Evidence: n/a

```
git config —local color.ui false
```

Step: 6

Confirm: Git is configured to disable color.

Expectation: .git/config ui section shows color set to true.

Command: echo cat .git/config

Test Result: PASS

Evidence: Starts on next line.

```
1 cat .git/config
```

Step: 7

Confirm: Confirm a developer is able to add file f1 to a local repo.

Expectation: No error.

Command: echo git add f1

Test Result: PASS

Evidence: n/a

```
1 git add f1
```

Step: 8

Confirm: Confirm a developer is able to add file f2 to a local repo.

Expectation: No error.

Command: echo git add f2

Test Result: PASS

Evidence: n/a

```
1 git add f2
```

Step: 9

Confirm: Confirm a developer is able to touch a file named f3.

Expectation: No error.

Command: echo touch f3

Test Result: PASS

Evidence: n/a

```
1 touch f3
```

Step: 10

Confirm: Confirm a developer is able to touch a file named f4.

Expectation: No error.

Command: echo touch f4

Test Result: PASS

Evidence: n/a

```
touch f4
```

Step: 11

Confirm: Confirm a developer has obtained a list of tracked and untracked repo changes.

Expectation: Status shows tracked files f1 and f2 and untracked files f3 and f4.

Command: echo git status

Test Result: PASS

Evidence: Starts on next line.

```
1 git status
```

Step: 12

Confirm: Confirm a developer is able to obtain a list of untracked files.

Expectation: Untracked files f3 and f4 are listed.

Command: echo git clean -n

Test Result: PASS

Evidence: Starts on next line.

```
1 git clean -n
```

Step: 13

Confirm: Confirm a developer is able to remove untracked files.

Expectation: Untracked files f3 and f4 are removed.

Command: echo git clean -f

Test Result: PASS

Evidence: Starts on next line.

```
1 git clean -f
```

Step: 14

Confirm: Confirm a developer is able to commit files f1 and f2 to the local repo.

Expectation: No errror.

Command: echo git commit -m "Initial-commit."

Test Result: PASS

Evidence: Starts on next line.

```
1 git commit -m Initial-commit.
```

Step: 15

Confirm: Confirm a developer is able determine tracked files have been committed to local repo.

Expectation: The working directory is clean.

Command: echo git status

Test Result: PASS

Evidence: Starts on next line.

```
1 git status
```

Step: 16

Confirm: Confirm a developer is able to list commits to local repo.

Expectation: Commit has been processed.

Command: echo git log –all

Test Result: PASS

Evidence: Starts on next line.

```
1 git log --all
```

5.1.5 Test Suite: Clone Remote Repository

Purpose: This test suite ensures a developer can clone a remote repository, make changes to it, and revert the changes.

Requirement: IUR-003, IUR-004, IUR-005, IUR-007, and IUR-008

5.1.6 Test Case: Clone Repository

Purpose: This test case demonstrates a developer is able to clone a remote repository and use it as a local repository.

Requirement: IUR-004 and IUR-007

Step: 1

Confirm: A developer is able to clone a remote repository.

Expectation: No error.

Command: echo git clone https:github.com/Traap/quicksort.git

Test Result: PASS

Evidence: Starts on next line.

```
1 git clone https:github.com/Traap/quicksort.git
```

Step: 2

Confirm: A developer is able to disable color output to the terminal.

Expectation: No error.

Command: echo git config –local color.ui false

Test Result: PASS

Evidence: n/a

```
git config —local color.ui false
```

Step: 3

Confirm: Git is configured to disable color.

Expectation: .git/config ui section shows color set to true.

Command: echo cat .git/config

Test Result: PASS

Evidence: Starts on next line.

```
cat .git/config
```

Step: 4

Confirm: A developer is working on master branch and up-to-date after cloning a remote repository.

Expectation: Branch information is displayed.

Command: echo git status

Test Result: PASS

Evidence: Starts on next line.

```
1 git status
```

Step: 5

Confirm: A developer is able to review a set of tracked repositories.

Expectation: Fetch and pull URLs are identical

Command: echo git remote -v

Test Result: PASS

Evidence: Starts on next line.

```
1 git remote -v
```

Step: 6

Confirm: A developer's local and remote branches are mapped to master branch.

Expectation: Local and remote branch are mapped to master branch..

Command: echo git branch -vv

Test Result: PASS

Evidence: Starts on next line.

```
1 git branch -vv
```

5.1.7 Test Case: Revert a merged change

Purpose: This test case demonstrates a developer is able to make a change to a local git repository, push the changes to a remote repository, and then revert the entire change.

Requirement: IUR-003, IUR-004, IUR-005, IUR-007, and IUR-008

Step: 1

Confirm: A developer is able to pull the remote repository to the local repository.

Expectation: No error.

Command: echo git pull

Test Result: PASS

Evidence: n/a

```
1 git pull
```

Step: 2

Confirm: A developer is able to remove all files from the local repository.

Expectation: No error.

Command: echo git rm src/Main.hs src/QuickSort.hs

Test Result: PASS

Evidence: n/a

```
1 git rm src/Main.hs src/QuickSort.hs
```

Step: 3

Confirm: A developer is able to commit file removals.

Expectation: No error.

Command: echo git commit -m "Remove all quicksort files."

Test Result: PASS

Evidence: Starts on next line.

```
1 git commit -m Remove all quicksort files.
```

Step: 4

Confirm: A developer is able to commit file removals.

Expectation: No error.

Command: echo git push

Test Result: PASS

Evidence: n/a

```
1 git push
```

Step: 5

Confirm: A developer is able to review the commits to the repository.

Expectation: Update has been processed.

Command: echo git log –oneline -3

Test Result: PASS

Evidence: Starts on next line.

```
git log ––oneline –3
```

Step: 6

Confirm: A developer is able to pull the remote repository to the local repository.

Expectation: No error.

Command: echo git pull

Test Result: PASS

Evidence: Starts on next line.

```
1 git pull
```

Step: 7

Confirm: A developer is able ot undo the delete operation.

Expectation: No error.

Command: echo git revert –no-edit HEAD

Test Result: PASS

Evidence: Starts on next line.

```
1 git revert ––no-edit HEAD
```

Step: 8

Confirm: A developer is albe to update the remote repository.

Expectation: No error.

Command: echo git push

Test Result: PASS

Evidence: n/a

```
1 git push
```

Step: 9

Confirm: A developer is able to review the commits to the repository.

Expectation: Update has been processed.

Command: echo git log –oneline -3

Test Result: PASS

Evidence: Starts on next line.

```
git log ––oneline –3
```

5.1.8 Test Suite: Quick Fix to Software

Purpose: This test suite ensures a developer is able to swich working contexts from the master branch to a quick-fix branch to make programming changes.

Requirement: IUR-002, IUR-003, IUR-004, IUR-005, IUR-006, IUR-007, IUR-008, and IUR-009.

5.1.9 Test Case: Create a branch named Quick Fix

Purpose: This test case demonstrates a developer is able to create a new branch to make a programming fix. The new branch is tracked by both the local and remote git repository. The developer is able to make a code change and commit the change to the local git repository and push it to the remote git repository.

Requirement: IUR-002, IUR-003, IUR-004, IUR-005, IUR-006, IUR-007, IUR-008, and IUR-009.

Step: 1

Confirm: A developer is able to download a git repository.

Expectation: No error.

Command: echo git pull

Test Result: PASS

Evidence: n/a

```
1 git pull
```

Step: 2

Confirm: A developer is able to create a new branch.

Expectation: No error.

Command: echo git checkout -b quick-fix

Test Result: PASS

Evidence: n/a

```
1 git checkout -b quick-fix
```

Step: 3

Confirm: A developer is able to obtain a listing of all branches.

Expectation: No error.

Command: echo git branch -a

Test Result: PASS

Evidence: Starts on next line.

```
1 git branch -a
```

Step: 4

Confirm: A developer is able to make a code change.

Expectation: No error.

Command: echo sed -i.bak -e s/\\[a\\]/\\[Z\\]/g QuickSort.hs

Test Result: PASS

Evidence: n/a

```
1 sed -i.bak -e s/\[a\]/\[Z\]/g QuickSort.hs
```

Step: 5

Confirm: A developer is able to remove untracked files.

Expectation: No error.

Command: echo git clean -f

Test Result: PASS

Evidence: Starts on next line.

```
1 git clean -f
```

Step: 6

Confirm: A developer is able to add the changed file to the local repository.

Expectation: No error.

Command: echo git add .

Test Result: PASS

Evidence: n/a

```
1 git add .
```

Step: 7

Confirm: A developer is able to determine the changed file has been added to the tracked changes on the local git repository.

Expectation: Tracked file is listed.

Command: echo git status

Test Result: PASS

Evidence: Starts on next line.

```
1 git status
```

Step: 8

Confirm: A developer is able to commit the changed file.

Expectation: No error.

Command: echo git commit -m "Change all [a] to [Z]"

Test Result: PASS

Evidence: Starts on next line.

```
1 git commit -m Change all [a] to [Z]
```

Step: 9

Confirm: A developer is able to update the remote repository.

Expectation: No error.

Command: echo git push origin quick-fix

Test Result: PASS

Evidence: n/a

```
1 git push origin quick-fix
```

Step: 10

Confirm: A developer is able to establish branch tracking between the remote repository and new branch.

Expectation: Tracking has been establishe.

Command: echo git branch -u origin/quick-fix quick-fix

Test Result: PASS

Evidence: Starts on next line.

```
1 git branch -u origin/quick-fix quick-fix
```

Step: 11

Confirm: A developer is able to update the remote repository.

Expectation: No error.

Command: echo git push

Test Result: PASS

Evidence: n/a

```
1 git push
```

Step: 12

Confirm: A developer is able to view difference between master and quick-fix branches.

Expectation: No error.

Command: echo git diff master..quick-fix

Test Result: PASS

Evidence: Starts on next line.

```
1 git diff master..quick-fix
```

Step: 13

Confirm: A developer is able to verify branches have been linked.

Expectation: Branch linkage is established.

Command: echo git branch -vv

Test Result: PASS

Evidence: Starts on next line.

```
1 git branch -vv
```

Step: 14

Confirm: A developer is able to review commits made to repository.

Expectation: Update has been processed.

Command: echo git log –oneline -3

Test Result: PASS

Evidence: Starts on next line.

```
git log ––oneline –3
```

5.1.10 Test Case: Undo a program change made on branch Quick Fix.

Purpose: This test case demonstrates a developers is able to checkout a branch named Quick Fix. The developer is able to reverse the programming change and commit the change to the local git repository and push it to the remote git repository.

Requirement: IUR-003, IUR-004, IUR-005, IUR-006, IUR-007, IUR-008, and IUR-009.

Step: 1

Confirm: A developer is able to switch to the master branch.

Expectation: No error.

Command: echo git checkout master

Test Result: PASS

Evidence: Starts on next line.

```
1 git checkout master
```

Step: 2

Confirm: A developer is able refresh their local repository after a remote repository has been downloaded.

Expectation: No error.

Command: echo git pull

Test Result: PASS

Evidence: Starts on next line.

```
1 git pull
```

Step: 3

Confirm: A developer is able to create a new branch to work on a programming assignment.

Expectation: No error.

Command: echo git checkout quick-fix

Test Result: PASS

Evidence: Starts on next line.

```
1 git checkout quick-fix
```

Step: 4

Confirm: A developer is able to establish branch tracking between the local and remote branches.

Expectation: No error.

Command: echo git branch -u origin/quick-fix quick-fix

Test Result: PASS

Evidence: Starts on next line.

```
1 git branch -u origin/quick-fix quick-fix
```

Step: 5

Confirm: A developer is able to list all branches.

Expectation: No error.

Command: echo git branch -a

Test Result: PASS

Evidence: Starts on next line.

```
1 git branch -a
```

Step: 6

Confirm: A developer is able to make a software change.

Expectation: No error.

Command: echo sed -i.bak -e s/\\[Z\\]/\\[a\\]/g QuickSort.hs

Test Result: PASS

Evidence: Starts on next line.

```
1 sed -i.bak -e s/\[Z\]/\[a\]/g QuickSort.hs
```

Step: 7

Confirm: A developer is able to remove files that are not being tracked by the local repository.

Expectation: No error.

Command: echo git clean -f

Test Result: PASS

Evidence: Starts on next line.

```
1 git clean -f
```

Step: 8

Confirm: A developer is able to add changed files to the local repository.

Expectation: No error.

Command: echo git add .

Test Result: PASS

Evidence: Starts on next line.

```
1 git add .
```

Step: 9

Confirm: A developer is able to determine thier change has been added to the local repository.

Expectation: File changes are being tracked by local repository.

Command: echo git status

Test Result: PASS

Evidence: Starts on next line.

```
1 git status
```

Step: 10

Confirm: A developer is able to commit change to the the local repository.

Expectation: No error.

Command: echo git commit -m "Change all [Z] to [a]"

Test Result: PASS

Evidence: Starts on next line.

```
1 git commit -m Change all [Z] to [a]
```

Step: 11

Confirm: A developer is able to update a remote Git repository based on changes made to the local repository.

Expectation: No error.

Command: echo git push

Test Result: PASS

Evidence: Starts on next line.

```
1 git push
```

Step: 12

Confirm: A developer is able to review the differences between their master branch and quick-fix branch.

Expectation: Differences are displayed.

Command: echo git diff master..quick-fix

Test Result: PASS

Evidence: Starts on next line.

```
1 git diff master..quick-fix
```

Step: 13

Confirm: A developer is able to determine their new branch is tracked independently of the master branch.

Expectation: Branch tracking is displayed.

Command: echo git branch -vv

Test Result: PASS

Evidence: Starts on next line.

```
1 git branch --vv
```

Step: 14

Confirm: A developer is able to review a list of changes made to their repository.

Expectation: Commits are displayed.

Command: echo git log –oneline -3

Test Result: PASS

Evidence: Starts on next line.

```
git log ––oneline –3
```

5.1.11 Test Case: Merge a programing change made on branch named Quick Fix

Purpose: This test case demonstrates a developer's ability to merge a code change from a working branch to the master branch that the team treats as their central repository makes the code available to others.

Requirement: IUR-002, IUR-003, IUR-004, IUR-006, IUR-007, IUR-008, and IUR-009.

Step: 1

Confirm: A developer is able to checkout the master branch to the local repository.

Expectation: No error.

Command: echo git checkout master

Test Result: PASS

Evidence: Starts on next line.

```
1 git checkout master
```

Step: 2

Confirm: A developer is able to update the local copy of their master branch.

Expectation: No error.

Command: echo git pull

Test Result: PASS

Evidence: n/a

```
1 git pull
```

Step: 3

Confirm: A developer is able to checkout a quick-fix branch to their local repository.

Expectation: No error.

Command: echo git checkout quick-fix

Test Result: PASS

Evidence: Starts on next line.

```
1 git checkout quick-fix
```

Step: 4

Confirm: A developer is able to update the local copy of their quick-bix branch.

Expectation: No error.

Command: echo git pull

Test Result: PASS

Evidence: n/a

```
1 git pull
```

Step: 5

Confirm: A developer is able to return to their master branch.

Expectation: No error.

Command: echo git checkout master

Test Result: PASS

Evidence: Starts on next line.

```
1 git checkout master
```

Step: 6

Confirm: A developer is able to review the differences between their master branch and quick-fix branch.

Expectation: Differences between master and quick-fix branch are listed.

Command: echo git diff master..quick-fix

Test Result: PASS

Evidence: Starts on next line.

```
1 git diff master..quick-fix
```

Step: 7

Confirm: A developer is able to determine the quick-fix branch has not been merged with the master branch.

Expectation: Quick-fix branch has not been merged.

Command: echo git branch –merged

Test Result: PASS

Evidence: Starts on next line.

```
git branch —-merged
```

Step: 8

Confirm: A developer is able to merge the quick-fix branch into the master branch.

Expectation: No error.

Command: echo git merge quick-fix

Test Result: PASS

Evidence: Starts on next line.

```
1 git merge quick-fix
```

Step: 9

Confirm: A developer is able to update the remote repository with the results of the merge operation.

Expectation: No error.

Command: echo git push

Test Result: PASS

Evidence: Starts on next line.

```
1 git push
```

Step: 10

Confirm: A developer is able to review the results of the merge operation.

Expectation: Quick-fix branch has been merged.

Command: echo git branch –merged

Test Result: PASS

Evidence: Starts on next line.

```
git branch ––merged
```

Step: 11

Confirm: A developer is able to delete the quick-fix branch from the local repository.

Expectation: Local quick-fix branch has been removed.

Command: echo git branch –delete quick-fix

Test Result: PASS

Evidence: Starts on next line.

```
1 git branch --delete quick-fix
```

Step: 12

Confirm: A developer is able to delete the quick-fix branch from the remote repository.

Expectation: No error.

Command: echo git push origin –delete quick-fix

Test Result: PASS

Evidence: Starts on next line.

```
1 git push origin —-delete quick-fix
```

Step: 13

Confirm: A developer is able to determine the remote and local quick-fix branches have been deleted.

Expectation: No error.

Command: echo git branch -vv

Test Result: PASS

Evidence: Starts on next line.

```
1 git branch -vv
```

Step: 14

Confirm: A developer is able to review a list of changes made to the repository.

Expectation: Commits are listed.

Command: echo git log –oneline -3

Test Result: PASS

Evidence: Starts on next line.

```
git log ––oneline –3
```

5.1.12 Test Suite: Git Tagging

Purpose: This test suite ensures a developer is able to tag git-commits, create new branches based on git-tags, and perform new work on a new branch.

Requirement: IUR-002, IUR-003, IUR-004, IUR-006, IUR-007, IUR-008, and IUR-009.

5.1.13 Test Case: Tag Commits

Purpose: This teset case is used to demonstrate Git's abbility to tag a commit. A Git repository has been pre-seeded with the following tags. {longtable} {{1.2cm}{10cm}|} Tag & Git Checksum\\v1.0 & bcde6bf1ed2841f39e719ccd323df4cc33497e04 \\v1.1 & b64d30b59ff026f12350a5c040d779a5a9a59214 \\v1.2 & 8aa0308a4b661e97df4002ccd7f0ee3dd2f559 \\v2.0 & ea06210d6f84c2daf26f00c39643a9cd9b98a4a5 \\v2.1 & 440ba8b56feb5941783b9cb57ed1923 \\v2.2 & c47279b87ed774137929088182l5f53364567bb1 \\{longtable} A Git Checksum is unique to a Git installation and are listed here as examples. In the event these tags do not exist, the following commands are used to recreate them and publish them to the git server. {git tag v1.0 [git-checksum]} where git-checksum is the git-commit-id you are associating the tags with. Git-commit-id's are globally-uniquie. {git push origin –tags}

Requirement: IUR-002, IUR-003, IUR-004, IUR-006, IUR-007, IUR-008, and IUR-009.

Step: 1

Confirm: Pre-existing tags are present.

Expectation: Pre-existing tags are listed without error.

Command: echo git tag

Test Result: PASS

Evidence: Starts on next line.

```
1 git tag
```

Step: 2

Confirm: Pre-existing git-commits are present.

Expectation: Git-commit history is shown without error.

Command: echo git log –graph –pretty=format:"%h (%cr) <%cn> %d" –simplify-by-decoration

Test Result: PASS

Evidence: Starts on next line.

```
1 git log --graph --pretty=format:%h (%cr) <%cn> %d --simplify-by-decoration
```

Step: 3

Confirm: A complete list of files and revision information for v2.0 is listed.

Expectation: All files associated with v2.0 are listed.

Command: echo git ls-tree –full-tree -r –name-only v2.0

Test Result: PASS

Evidence: Starts on next line.

```
git ls-tree --full-tree -r --name-only v2.0
```

Step: 4

Confirm: New branch b2.0 is created from tag v2.0.

Expectation: Switch to a new branch "b2.0" is displayed.

Command: echo git checkout -b b2.0 v2.0

Test Result: PASS

Evidence: Starts on next line.

```
1 git checkout -b b2.0 v2.0
```

Step: 5

Confirm: New branch b1.1 is created from tag v1.1.

Expectation: Switch to a new branch "b1.1" is displayed.

Command: echo git checkout -b b1.1 v1.1

Test Result: PASS

Evidence: Starts on next line.

```
1 git checkout –b b1.1 v1.1
```

Step: 6

Confirm: (HEAD, tag: v1.1, b1.1) is displayed for first git-commit.

Expectation: Git-commit history shows (HEAD, tag: v1.1, b1.1) at first commit.

Command: echo git log –graph –pretty=format:"%h (%cr) <%cn> %d" –simplify-by-decoration

Test Result: PASS

Evidence: Starts on next line.

```
1 git log --graph --pretty=format:%h (%cr) <%cn> %d --simplify-by-decoration
```

Step: 7

Confirm: Branches b1.1, b2.0 and master exist.

Expectation: Branches b1.1, b2.0 and master are listed. More branches is okay.

Command: echo git branch -v

Test Result: PASS

Evidence: Starts on next line.

```
1 git branch -v
```

Step: 8

Confirm: Switch to master branch.

Expectation: Switch to branch "master" is displayed.

Command: echo git checkout master

Test Result: PASS

Evidence: Starts on next line.

```
1 git checkout master
```

Step: 9

Confirm: Master branch is not tagged with v3.0.

Expectation: At minimum, "(HEAD, origin/master, origin/HEAD, master)" is displayed for first commit.

Command: echo git log –graph –pretty=format:"%h (%cr) <%cn> %d" –simplify-by-decoration

Test Result: PASS

Evidence: Starts on next line.

```
git log --graph --pretty=format:%h (%cr) <%cn> %d --simplify-by-decoration
```

Step: 10

Confirm: Add tag "v3.0" is added to HEAD.

Expectation: No error.

Command: echo git tag v3.0

Test Result: PASS

Evidence: Starts on next line.

```
1 git tag v3.0
```

Step: 11

Confirm: Tag "v3.0" is added to HEAD.

Expectation: At minimum, '(HEAD, tag: v3.0, origin/master, origin/HEAD, master)' is displayed for first commit.

Command: echo git log –graph –pretty=format:"%h (%cr) <%cn> %d" –simplify-by-decoration

Test Result: PASS

Evidence: Starts on next line.

```
git log --graph --pretty=format:%h (%cr) <%cn> %d --simplify-by-decoration
```

Step: 12

Confirm: Tag "v3.0" is deleted.

Expectation: No error.

Command: echo git tag -d v3.0

Test Result: PASS

Evidence: Starts on next line.

```
1 git tag –d v3.0
```

Step: 13

Confirm: Branch "b1.1" is deleted.

Expectation: No error.

Command: echo git branch -d b1.1

Test Result: PASS

Evidence: Starts on next line.

```
1 git branch -d b1.1
```

Step: 14

Confirm: Branch "b2.0" is deleted.

Expectation: No error.

Command: echo git branch -d b2.0

Test Result: PASS

Evidence: Starts on next line.

```
1 git branch -d b2.0
```

Step: 15

Confirm: Master branch remains.

Expectation: No error.

Command: echo git branch -v

Test Result: PASS

Evidence: Starts on next line.

```
1 git branch -v
```

6 Configuration Item Conclusion

This Report Package has satisfied the IUR for the Configuration Item described herein thus the Configuration Item is considered validated for its intended use.

Change Summary

Change	Justification
v1.0.0 Initial version.	New document.
v1.0.1 1. Corrected gramitical errors throughout document. 2. Added test case to demonstrate advanced git branching and tagging.	Process engineering feedback from Bitbucket pull-request