

Department of Veterans Affairs

Open Source Electronic Health Record Services

Roll and Scroll Recorder Software Design Document



**Version 0.5
January 2013**

Revision History

Date	Revision	Description	Author
12/29/2012	0.1	Initial Draft	Jimmy Spivey
01/02/2013	0.2	Peer Review	Meredith Watkins
01/02/2013	0.3	Peer Review	Kathleen Keating
01/10/2013	0.4	Updated Revision	Jimmy Spivey
02/08/2013	0.5	Changes for A/V detection and mutli-select	Jimmy Spivey

Table of Contents

1.	Introduction.....	7
1.1.	Roll-and-Scroll Recorder (RASR).....	7
1.1.1.	Design Overview	7
2.	Leveraging an Existing Open Source SSH Terminal.....	8
2.1.	Current Status of JCTerm.....	8
2.2.	JCTerm Class Diagram	8
2.3.	JCTerm Sequence Diagrams.....	10
2.3.1.	Establishing an SSH Connection	10
2.4.	Modification of JCTerm to Create RASR	10
3.	Implementation Specs of RASR Features.....	11
3.1.	Terminal Session Recording	11
3.2.	Preventing Users Access and Verify Codes From Being Recorded	11
3.3.	Expected Value Override	12
3.4.	Executing Recorded Tests	12
3.4.1.	Running via the Lightweight Test Harness	13
3.4.1.1.	Lightweight Test Harness Specification	13
3.4.2.	Linking to a Test Harness	14
3.4.3.	Interfacing to the OSEHRA Automated Testing Framework (OAT) ...	14
3.4.3.1.	Adhering to the OAT's Standards	14
3.4.3.2.	Modifications to the OAT to Support RASR's Features	14
3.4.4.	New Python Code Structure	14
3.4.4.1.	File Naming Convention for Python Test Scripts.....	17
3.4.4.2.	Test Suite Configuration Files	17
3.4.4.3.	Local User Configuration Files	18
4.	RASR Class Diagrams.....	19
5.	RASR Sequence Diagrams	21
5.1.	RASR Plug-in and View Initialization	21
6.	User Interface.....	21
7.	Appendix	21
7.1.	Acronyms and Definitions	21
7.2.	Software Licenses	22
7.2.1.	Software under License	22
7.2.2.	License Locations	23

7.3.	Document References.....	24
-------------	---------------------------------	-----------

1. Introduction

The Department of Veterans Affairs (VA) has contributed the latest U.S. Department of State Freedom of Information Act (FOIA) release of the Veterans Health Information Systems and Technology Architecture (VistA) codebase to Open Source Electronic Health Record Agent (OSEHRA), the custodial agent that serves as the central governing body of a new open source community. The Open Source Electronic Health Record (EHR) Services project includes VistA Data Comparison, VistA System Test Platform, VistA Refactoring, VistA System Test Scripts, Veterans Benefits Administration (VBA) System Test Platform, Eclipse Plug-In Tool, and VistA Meaningful Use Certification.

1.1. Roll-and-Scroll Recorder (RASR)

All Automated Testing Framework (ATF) test scripts are written in Python. The framework provides an extensible reference model from which additional tests can be constructed. Test developers do not require extensive programming skills to utilize the framework.

However, there are a number of software quality assurance (SQA) personnel at the VA who have a wealth of test knowledge, and are skilled in testing VistA through the roll-and-scroll interface.

To leverage the skills of these SQA resources, a roll-and-scroll recording feature is necessary. Using the recording feature, a SQA resource will execute a test manually and thus create a record of a given test sequence. The actions and expected responses will be automatically recorded and saved as a test sequence (i.e. Python script) within the test tool. The Python script can be edited if necessary. All tests can be re-executed automatically, as individual tests, or as a suite of tests from within the framework.

1.1.1. Design Overview

The Roll-and-Scroll Recorder (RASR) has been developed by enhancing an existing Eclipse SSH Plug-in called JCTerm Plug-in. The JCTerm Eclipse Plug-in can establish SSH connections and render interactive text based interfaces, such as VistA. New features such as terminal session recording and the ability to export sessions as Python test files to the OSEHRA Automated Test Framework (OAT) have been added. This Design documents will go over the JCTerm itself, the changes made to it to create RASR, as well as new features added to RASR.

2. Leveraging an Existing Open Source SSH Terminal

JCTerm is a stand-alone, open source application developed in Java. It contains a VT-100 terminal emulator. VT-100 terminal emulation is compatible with connecting and interfacing to VistA. JCTerm uses Java's Abstract Window Toolkit (AWT) and uses the AWT class `java.awt.image.BufferedImage` to render its VT-100 terminal emulation. JCTerm also exists as an Eclipse plug-in. It works by creating a new view which contains JCTerm inside of it. Since Eclipse plug-ins rely on Standard Widget Toolkit (SWT) components framework, `org.eclipse.swt.awt.SWT_AWT.new_Frame(Composite parent)` is used as a bridge to allow AWT components to exist inside of Eclipse as SWT wrappers. This effectively allows JCTerm to just be placed directly into Eclipse without change.

The RASR project is a fork of the JCTerm Eclipse Plug-in project. Therefore the JCTerm source code the RASR uses is licensed under the Eclipse Public License v 1.0 (EPL 1.0), whereas JCTerm (the stand alone Java application) is licensed under GNU Library GPL Version 2, which is not to be confused with the acronym GNU LGPL, which stands for GNU Lesser Public License.

2.1. Current Status of JCTerm

The RASR was created by importing the JCTerm Eclipse Plug-in version 0.0.2, which internally contains JCTerm version 0.0.9. This plug-in has not been updated in the past nine (9) months.

The JCTerm plug-in could benefit from the following work:

- Finishing the VT-100 Terminal emulation, as noted on JCTerm Plug-in's website.
- Rendering issues with the AWT `BufferedImage` class (one consideration for this: using a Rich Text editor instead of a rendered image for the terminal display)

2.2. JCTerm Class Diagram

The class diagram in Figure 1 is for the original, unmodified JCTerm Eclipse plug-in version 0.0.2. The classes `JCTermSWT.java` and `JCTermSwing.java` were omitted since they are not used.

Furthermore, this class diagram only shows a relationship between two classes if one of the classes contains an instance field of the other class. So if a class instantiates another class but does not store it as an instance field (a variable outside of a method/inside the class), it will not be mapped on this class diagram.

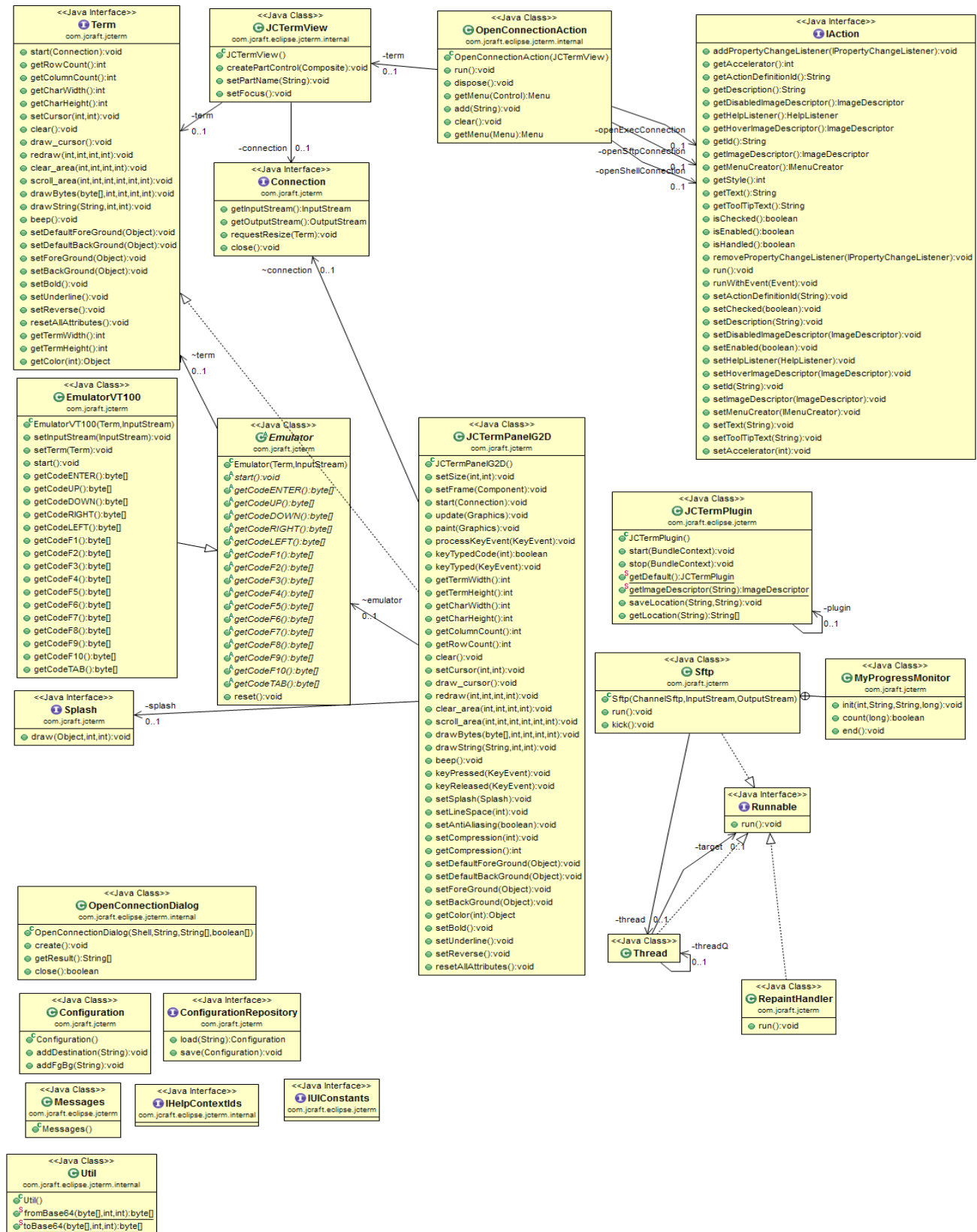


Figure 1: Class Diagram of Unmodified JCTerm Eclipse Plug-in
Please refer to JCTermPluginClassDiagram.png for full size.

2.3. JCTerm Sequence Diagrams

2.3.1. Establishing an SSH Connection

There is no need to modify this portion of the JCTerm code as it is already an established SSH2 solution. So it is simply leveraged, as-is, in the RASR.

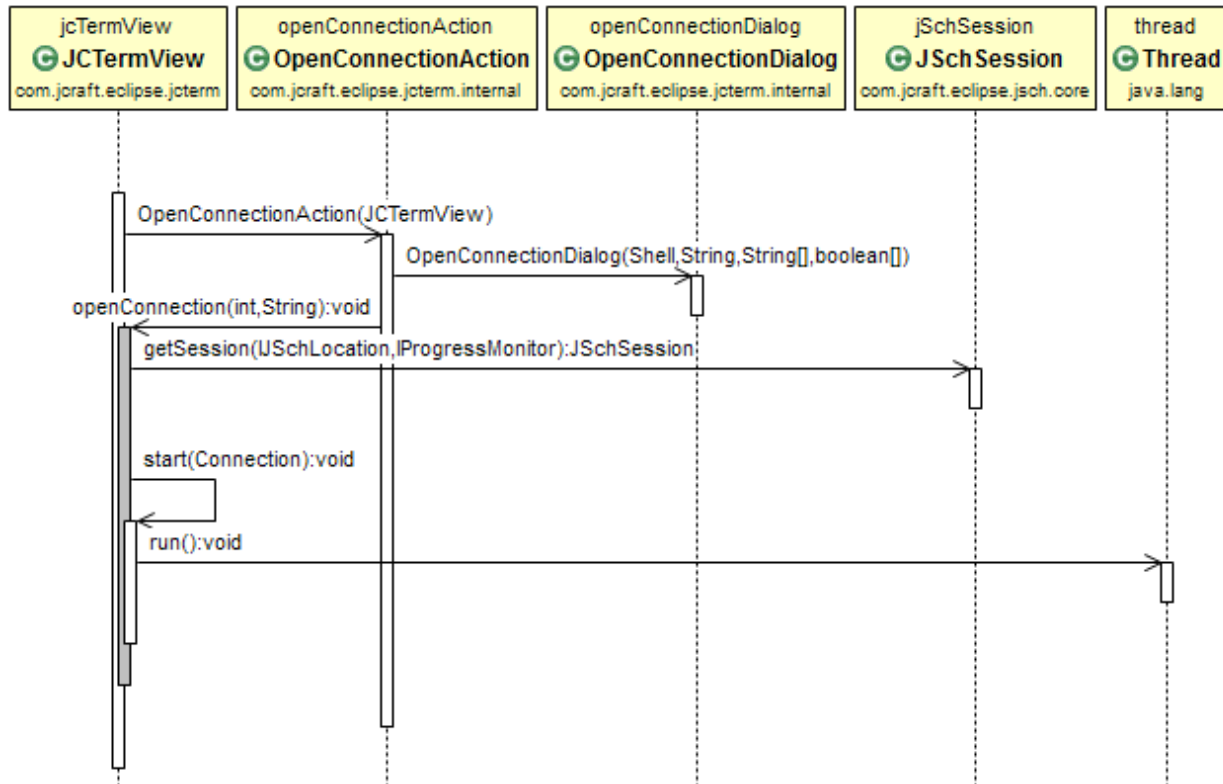


Figure 2: Sequence Diagram of Unmodified JCTerm's SSH Connection

2.4. Modification of JCTerm to Create RASR

JCTerm has been modified by bringing the plug-in into Eclipse as an Eclipse Plug-in Fragment. New features are added by either creating new Java class files, which go into a package which is outside of `com.jcraft`. The original JCTerm classes `JCTermPlugin.java` and `JCTermView.java` are also often modified in order to implement RASR's features. Since these features are specific to RASR they are not intended to be contributed back to the original JCTerm project. The major class `JCTermAWT.java` was renamed to `ATFRecorderAWT.java`. Several unneeded classes were deleted related to Secure File Transfer Protocol (SFTP) as well as the unused Swing and SWT JCTerm implementations.

All changes to JCTerm have been tracked in GitHub, at <https://github.com/OSEHR/RASR>. Note in the URL the missing A from OSEHRA; there are two (2) OSEHRA repositories on GitHub. OSEHRA is the main repository and OSEHR is a sandbox repository.

3. Implementation Specs of RASR Features

3.1. Terminal Session Recording

The RASR records all of the terminal output the user enters and only some of the terminal input (i.e. a Vista screen). As opposed to capturing the entire screen, the last twenty (20) characters of the terminal input (minus any of the user's echoed keystrokes) are recorded by default. The user can override the default by specifying a specific set of characters to be recorded. The maximum amount that can be recorded, for a screen, is a one (1) line.

The algorithm used for dividing the terminal data into a screen is solely based on when the user presses the enter key. When the user presses enter, all of the recorded user keystrokes for this screen are captured and the characters selected in the Expected Value Eclipse View are captured. These two (2) strings are stored into the Java type `org.osehra.eclipse.atfrecorder.TestRecording`. This contains List of type `org.osehra.eclipse.atfrecorder.RecordableEvent`, which contains a single string and an ENUM indicating whether it is an EXPECT or SEND type. EXPECT is what comes in on the terminal screen (typically only the last 20 are saved) and SEND are the user keystrokes typed into the terminal session for that screen. This list remains active until it is ready to be exported as a python script file.

3.2. Preventing Users Access and Verify Codes From Being Recorded

To prevent VistA Access and Verify codes from being recorded, an algorithm detects the prompt for an Access and Verify code. If an access and verify code prompt is found, neither the users access code nor verify code are stored into the generated test script. Instead these values are stored into the user's local configuration file, and not into the OAT. The start and stop icons are also disabled during A/V prompts to indicate that there is no recording currently.

Access and Verify code prompts are detected based on the current screen's contents. If the first screen ends with "ACCESS CODE:" and the subsequent screen only contains "VERIFY CODE:" then an A/V code prompt is detected.

By preventing Access and Verify codes from being recorded, test script files can be shared amongst others testers. The user must later configure their own access and verify code in their local user folder. This configuration file is located in `~/.ATF/roles.cfg`.

3.3. Expected Value Override

A new Eclipse view called "Expected Value" reflects the current terminal screen. This view is displayed side-by-side along with the JCTerm terminal. This view allows the user to select (override) the expected value recorded from each screen. Why a separate view? The expected value selection process cannot be performed in the JCTerm view because JCTerm uses a rendered image.

This new view, which is a duplicate of the current terminal input, is selectable text. In this view, the last twenty (20) characters, of the last line, are selected and highlighted. The user may override this value by selecting any number of characters on any line in the screen.

3.3.1. Selecting Multiple Values

For a given screen multiple values can be selected, as opposed to one. This will generate multiple expect statements (e.g. `wait("selected value 1") wait("selected value 2") ...`). Because the Expected Value View is a large text area, using the operating systems standards for selecting text, it only allows 1 segment of text to be selected at any time. Therefore the user will have a mutli-select button to allow them to select text segments in the current screen.

When the mutli-select value button is pressed, it will be depressed. If pressed again it will leave mutli-select mode, and also ignore any previously selected text. When the enter key is pressed, the user moves onto the next screen as usual. Each selected value will generate a wait statement and the mutli-select value button will be toggled off, returning to normal single value select mode.

The user will need to see what is currently selected as the list builds. This shall be accomplished in a text box which is below the text area. In this text box, a list of values shall be displayed in the format of "value1", "value2".

3.4. Executing Recorded Tests

The user must specify a test harness, a location in the file system, into which the Python scripts shall be saved. This may be a local instance of a OSEHRA-Automated-Testing repository/project, or a *lightweight* test harness. Since the Python scripts have dependencies, the test harness will be setup and configured to run the Python test files, as well as make supplying parameters via command line interface or configuration files easy. In other words, the RASR only exports the Python test scripts. The test harness is responsible for handling connections, providing the required libraries, and reporting tests as passed or failed.

3.4.1. Running via the Lightweight Test Harness

The user may wish to just setup a minimal test harness instead of the OAT. This test harness only contains the necessary files for running the tests the RASR exports.

3.4.1.1. Lightweight Test Harness Specification

The lightweight test harness shall contain only the features needed to be able to run the python tests which RASR generates. This shall be an Eclipse project, disturbed and managed in a publicly accessible VCS repository.

The lightweight test harness shall have the following requirements only:

Dependencies which the Eclipse project provides, copied from the OAT

- TestHelper.py
- RemoteConnection.py
- OSEHRAHelper.py

These required files are the very minimum needed for the lightweight test harness. These files are a subset of what is inside the OAT.

New items added to the lightweight test harness.

- A results directory.
- A packages directory which contains generated tests. It is identical to the FunctionalTest/RAS/VistA-FOIA/Packages directory, but instead located in the root of the eclipse project.
- A premade, and editable, Windows INI configuration file which provides what CMake-gui configuration items are needed: results directory, logging level, instance and namespace.
- A python file named RunAllTests.py. As the name suggests, it invokes all the tests in the packages directory.

These items allow for tests to be ran from a single file, as well as providing a simple, premade solution for configuring the tests so that they can be ran immediately after generating in RASR.

Dependencies which the machine must have installed

- Paramiko
- PyCrypto libraries (if on windows)

Whether running a RASR generated test in the OAT or the lightweight test harness, any machine will need to have these libraries installed.

3.4.2. Linking to a Test Harness

The user will be prompted to specify a file path to either the OAT or the lightweight test harness. This prompt is displayed in the event there is no linked test harness, since it is required.

3.4.3. Interfacing to the OSEHRA Automated Testing Framework (OAT)

The RASR exports recorded terminal sessions as Python script files into the OAT. This feature requires the RASR to understand some of the OAT's directory structure, its python, and CMake structures.

3.4.3.1. Adhering to the OAT's Standards

Since the RASR is linked to the OAT, RASR must understand OAT's directory structure, how OAT uses CMake, as well as Python libraries that the OAT uses. The preferences Java classes allow the user to link an OAT to the RASR. A Validation is performed inside the preference class to verify that the chosen directory is a compliant OAT directory. The Save Test button, which is calls the SaveTestAction and SaveTestDialog classes, save the exported python file into the directory [OAT Location] + /FunctionalTest/RAS/VistA-FOIA/Packages/. The Save Test button also invokes a background system process on the machine which calls CMake so that the OAT is rebuilt in case a new test suite is added.

3.4.3.2. Modifications to the OAT to Support RASR's Features

Some additional features and standards have been added to the OAT so that the RASR may export tests. These are:

- SSH support (which was to be added anyway)
- A new Python code structure for tests
- Naming tests in the FunctionalTest directory to the RASR standard
- Configuration files for test suites
- Configuration files specific to a local use

3.4.4. New Python Code Structure

Existing OAT should be updated to the new structure. This allows RASR to add new tests to existing test suites. It also brings in new features to the OAT such as SSH support and configuration files. The new Python code structure is backwards compatible. Existing functional test code does not need to be updated unless it wants to take advantage of the new SSH support. The new RASR-generated Python test scripts call into a global function in a separate Python file (TestHelper.py) at various stages of test execution. These functions (ie: `pre_test_suite_run`, invoked before any test suite runs) may be modified, thus allowing new python code to execute at various stages of test execution. These changes will apply to all previous tests without requiring any changes to those test's files. For example, we may wish to add or enhance the logging details of the OAT. We could modify one of the methods in TestHelper.py to add new logging statements, instead of editing all of the exported python

scripts. Figures 3 and 4 are sample Python driver and test suite files as per the new Python structure.

```

import os
import sys
#apparently these are not needed... at least not on windows. Will need to retest this
on linux
#sys.path = ['./FunctionalTest/RAS/lib'] + ['./lib/vista'] + sys.path
#sys.path = ['./'] + ['./lib/vista'] + sys.path

import ssh_connect_demo_suite
import TestHelper

def main():
    test_suite_driver = TestHelper.TestSuiteDriver(__file__)
    test_suite_details = test_suite_driver.generate_test_suite_details()

    try:
        test_suite_driver.pre_test_suite_run(test_suite_details)

        #Begin Tests
        ssh_connect_demo_suite.dive_into_menus(test_suite_details)
        #ssh_connect_demo_suite.demo_screen_man(test_suite_details)
        #End Tests

        test_suite_driver.post_test_suite_run(test_suite_details)
    except Exception, e:
        test_suite_driver.exception_handling(test_suite_details, e)
    else:
        test_suite_driver.try_else_handling(test_suite_details)
    finally:
        test_suite_driver.finally_handling(test_suite_details)

    test_suite_driver.end_method_handling(test_suite_details)

if __name__ == '__main__':
    main()

```

Figure 3: Sample Python Test Driver File

```

import sys
sys.path = ['./FunctionalTest/RAS/Lib'] + ['./dataFiles'] + ['./Lib/vista'] +
sys.path

from RActions import RActions
import TestHelper

def dive_into_menus(test_suite_details):
    testname = sys._getframe().f_code.co_name
    test_driver = TestHelper.TestDriver(testname)

    test_driver.pre_test_run(test_suite_details)
    try:
        vista = test_driver.connect_VistA(test_suite_details)
        rc = RActions(vista,
                      user=TestHelper.fetch_access_code(test_suite_details,
testname),
                      code=TestHelper.fetch_verify_code(test_suite_details,
testname))
        rc.signon()
        vista.wait('Select Training Menu Option:')
        vista.write('OE')
        vista.write('^')
        vista.wait(':')
        rc.signoff()

        test_driver.post_test_run(test_suite_details)
    except TestHelper.TestError, e:
        test_driver.exception_handling(test_suite_details, e)
    else:
        test_driver.try_else_handling(test_suite_details)
    finally:
        test_driver.finally_handling(test_suite_details)
        test_driver.end_method_handling(test_suite_details)

```

Figure 4: Sample Python Test Suite

3.4.4.1. File Naming Convention for Python Test Scripts

The RASR will require that Python test scripts all have the same file naming convention. This is so that the RASR can open, append, and create similarly named tests. Additionally, CMake/CTest will need a standard naming convention for executing tests (as opposed to executing non-test and supporting script files that the RASR or the user creates).

The RASR will create or edit three (3) Python script files within the specified subdirectory in the /Packages folder, a driver file (which is invoked by CTest), a test suite file which contains all of the test logic, and a configuration file. The driver file is named [test_suite_name]_test.py, the test suite file is named [test_suite_name]_suite.py, and the configuration file is named [test_suite_name].cfg. All of these files go into the directory /FunctionalTest/RAS/VistA-FOIA/Packages. Figure 5 contains correctly named files.

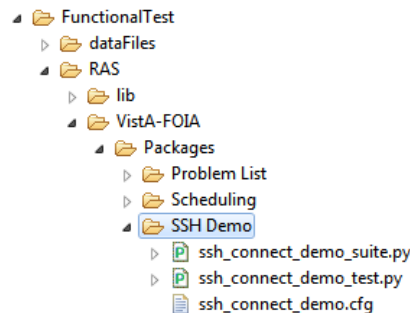


Figure 5: Correctly Named Test Files

3.4.4.2. Test Suite Configuration Files

Test suite configuration files are a new feature for the OAT; these are Windows INI files. These are created by the RASR, however configuring the fields may be required if the RASR generated defaults are incorrect (e.g. Instance or Namespace). The RASR will create a default configuration file in the naming format of [test_suite_name].cfg, as seen in Figure 5 above. RASR will prepopulate the file giving RemoteConnect a default value of 1, and populating the ServerLocation value with the one used by RASR to connect to the remote system. Sample contents, with values as they would be generated by RASR, of the file Test Suite Configuration file in Figure 6.

```
[RemoteDetails]
; If this is true, connect to a remote vista Instance via SSH
; If this false, connect locally to VistA-FOIA based on the local OS and ignore these
properties
RemoteConnect=1
ServerLocation=vista.vainnovation.us
Instance=cache
UseDefaultNamespace=1 ; Whether to send a ZN "[Namespace]" or not.
Namespace= ; Leave empty since not using
```

Figure 6: Sample Test Suite Configuration File

RemoteConnect=1 enables the SSH feature and then uses the address in ServerLocation field to connect. UseDefaultNamespace and Namespace are configuration fields which handle how to connect to VistA for instances where there are not default namespaces on the connected server. Instance is used to specify whether it connects to Cache or GTM, acceptable values are either cache or gtm.

3.4.4.3.Local User Configuration Files

Local user files are created on the user's machine in their home directory. The home directory for Linux is `~/` and `%UserProfile%` on Windows. Since Java is cross platform operable, this implementation detail is handled by default with Java. The configuration file produced by RASR is in the Windows INI format. By storing the configuration file into the tester's home directory, it allows tests scripts to be shared amongst other testers without sharing SSH usernames, passwords, VistA Access, or Verify codes. The configuration file contains values for the SSH username, password, and Access/Verify codes.

This file is placed into `~/.ATF/roles.cfg`. The RASR generates the configuration file, as well as populates all values. The user must set this up to connect tests. Figure 7 is a sample configuration file with two (2) separate test packages, each containing a single test suite and a test suite containing two (2) tests (Python functions):

```
[Problem List-PLMain01]
SSHUsername= MySSHUsername
SSHPassword= MySSHPassword
Startmon_aCode= ACCCESS1
Startmon_vCode= VERIFY1
pl_test001_aCode= ACCCESS1
pl_test001_vCode= VERIFY1

[SSH Demo-ssh_connect_demo]
SSHUsername=MySSHUsername
SSHPassword=MySSHPassword
dive_into_menus_aCode= ACCCESS1
dive_into_menus_vCode=SampleVerifyCode
demo_screen_man_aCode= ACCCESS2
demo_screen_man_vCode=SampleVerifyCode
```

Figure 7: Sample Local User Configuration File

It may be best to not have two separate types of configuration files for running tests. Any files stored inside the OAT file structure will likely be versioned in a Version Control System (VCS). If multiple users share configuration files, it will not allow each user to have their own unique values of `ServerLocation` for example. If a user synchronizes their source files with what is in the version control repository, they may step on other users by changing the configuration of a test suite. If this is a real problem, then it may be better to move all of the configuration to the local user directory since versioning control systems may step on other users who update shared configuration files.

Additionally, if an update needs to be made, it must be done to two (2) configuration files, which is not intuitive. In that case, a front end user GUI, which can house the logic to manipulate all the required files, as well as encrypt their contents would be benefit this latter scenario.

The root of the issue comes down to what configuration items should be shareable, so that they do not have to be reconfigured again when handed from one tester to another. In this case, they could be stored in VCS, but the configuration values must also be immutable as changes will impact other testers sharing the same version repository.

4. RASR Class Diagrams

As with any class diagram, only instance members (variables defined inside the class, not local/method members) will be linked. As such, some of the class diagrams here form separate

Open Source EHR Services

RASR System Design Document



January 2013

Figure 8: Class Diagram of RASR
Please refer to *RASRClassDiagram.png* for full size.

5. RASR Sequence Diagrams

Sequence diagrams for terminal emulation, session recording, and saving will be added in a later version of this document as implementation changes are currently pending.

5.1. RASR Plug-in and View Initialization

Figure 9 below is the sequence diagram for main view, the one which has been taken from JCTerm and enhanced. It also instantiates all the other Actions such as the original JCTerm OpenConnection, SaveTestAction, and PreferencesAction.

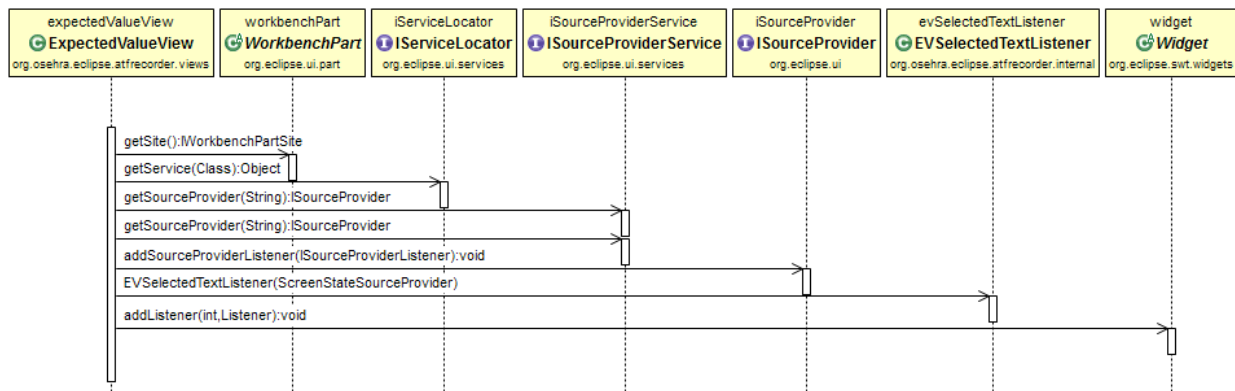


Figure 9: Main View Sequence Diagram

6. User Interface

Please refer to the RASR Usage and Installation document for user interface specifications.

7. Appendix

7.1. Acronyms and Definitions

AWT	Abstract Window Toolkit. A Java package for creating and managing Graphical user interfaces.
branch	Exists in a repository. Contains a set of revisions in a chronological order. There may be multiple branches in a repository, tracking the same files in parallel. These branches may later be merged into the main branch.
Eclipse	An IDE primarily used for Java software development.
Eclipse View	A tab inside of Eclipse which provides application features to aid in software development. Eclipse provides many by default (e.g. search, directory explorer and console). RASR and JCTerm Plug-in provide their functionality inside of their own Eclipse Views.

Eclipse Plug-in	An extension to the Eclipse application, which can be installed. It gives Eclipse new features for software development.
EPL	Eclipse Public License
fork	A copy of source code from one software project which creates a new separate project. Unlike a branch, there is no absolutely no intention of merging this back into its parent. Additionally, unlike a branch, it is a new project with new goals.
CMake	An application which configures and builds software. It takes source code and configuration files, as well as GUI input to produce software.
CTest	A testing tool which is part of CMake. It is used to invoke the OAT and run all of the automated tests.
GNU Library GPL	GNU Library General Public License
IDE	Integrated Developer Environment. A robust, text editing application which allows software developers to write and test code.
JCraft	A company who has contributed open source technology, namely JCTerm and JSch.
JCTerm	A Java application that can establish SSH connections and also provide terminal emulation. http://www.jcraft.com/jcterm/
JCTerm Plug-in	An Eclipse plugin that supports SSH and terminal emulation. It basically just JCTerm itself inside of an Eclipse View. http://www.jcraft.com/eclipse-jcterm/
JSch	An SSH module developed in Java. Used inside of JCTerm to establish SSH connections. http://www.jcraft.com/jsch/
Open Source	Software which is licensed under an open source license. This typically allows unrestricted modification and distribution of such licensed software.
OAT	OSEHRA Automated Testing Framework. A python based testing framework for testing the VistA application.
Paramiko	A Python SSH library
RASR	Roll and Scroll Recorder
SSH2	Secure Shell version 2, a cryptographic network protocol for secure communication. Often simply referred to as just SSH, since the original SSH-1 protocol is mostly defunct.
Software revision	A set of changes made to a software's source code. One or more (typically the latter) revisions make up a software version.
Terminal Emulator	An application that renders text-based user interfaces and accepts input from a command line. No graphics, only text is supported.
Version Control System	An application which manages all revisions and branches of revisions for a software project.
VistA	Veterans Health Information Systems and Technology Architecture

7.2. Software Licenses

7.2.1. Software under License

JCTerm	GNU Library GPL Version 2
JCTerm Plug-in	Eclipse Public License v 1.0
JSch	A custom "BSD-style license"
OAT	Apache License, Version 2.0
RASR	Apache License, Version 2.0

7.2.2. License Locations

GNU Library GPL Version 2	http://www.gnu.org/licenses/old-licenses/lgpl-2.0.html
Eclipse Public License v 1.0	http://www.eclipse.org/legal/epl-v10.html
JSch's custom license	http://www.jcraft.com/jsch/LICENSE.txt
Apache License, Version 2.0	http://www.apache.org/licenses/LICENSE-2.0.html

7.3. Document References