Development note

Contents

[Ziyuan 2](#_Toc433982998)

[Set up 2](#_Toc433982999)

[Build and deploy 2](#_Toc433983000)

[Structure 2](#_Toc433983001)

[TestData 3](#_Toc433983002)

[Special Components 3](#_Toc433983003)

[JunitRunner 3](#_Toc433983004)

[Dependencies 4](#_Toc433983005)

[Java Slicer 4](#_Toc433983006)

[Wala slicer 5](#_Toc433983007)

[JaCoCo 5](#_Toc433983008)

[JPDA(Java Platform Debugger Architecture) 5](#_Toc433983009)

[JavaIlp 6](#_Toc433983010)

[Slf4j (Simple Logging Facade for Java) 6](#_Toc433983011)

[Maven: 7](#_Toc433983012)

[Nexus server 7](#_Toc433983013)

[Deploy a library to nexus server 7](#_Toc433983014)

[Deploy a third-party library: 8](#_Toc433983015)

[Deploy project to the server 8](#_Toc433983016)

[Pom.xml 8](#_Toc433983017)

[Use classes in test folder of another module 8](#_Toc433983018)

# Ziyuan

## Set up

* Read [/etc/doc/Ziyuan\_Installation\_Note.txt](Ziyuan_Installation_Note.txt).

## Build and deploy

* In Eclipse: all common launching settings are stored in /app/tzuyu.parent/launches, these settings will be launched and added automatically into Run Configuration of Eclipse once tzuyu.parent is imported into Eclipse.
* To build the whole project, run parent build skip test:
  + In Eclipse, click Run → Run Configurations → Maven Build → parent build skip test→ click Run
  + There are also some launches to build a single module, such as sav.commons.launch, slicer.javaslicer.launch,…
* To deploy project onto nexus server:
  + Make sure that your maven settings already set username/password to access repositories on nexus server.
  + In Eclipse, run parent deploy (in Run Configuration)

## Structure

sav.commons

...

mutation

svm

Codecoverage.jacoco

Slicer.javaslicer

Tzuyu.core

* sav.commons: contains common utilities, models and interfaces for common services (slicer, code coverage…)
* slicer.javaslicer, codecoverage.jacoco, svm,… : each module provides a specific service, or an implementation/wrapper of an algorithm. These modules should depend on sav.commons and should not depend on other algorithm module directly, but using service interface provided in sav.commons instead.

This design is for avoiding tight coupling purpose.

* Tzuyu.core: the main idea is implemented in this module, where we combine all needed service together. Implementation of other ideas should also be in a separate module like this.

## TestData

This part is to explain how we test Ziyuan on real world projects.

Configurations:

* All parameters of testing project are defined in /etc/testdata.csv in which each column corresponds to a type defined in TestDataColumn enum.

Ex:

PROJECT\_NAME: testing project name, and also the project folder in testdata workspace.

BUG\_NUMBER: id of the issue (the bug of the testing project we are working on).

SOURCE\_FOLDER: relative path of source folder from project folder.

* For a cell with multiple values like classpath, values are separated by a semicolon (;)
* A cell begins with "#" will be ignored, and treated as an empty value.
* Other properties for loading testdata are in /etc/testdata.properties, such as testdata.workspace: the folder in which all testing projects are located.
* Added testcase for an issue has the prefix “TestIssue” or “Issue”.
* Testing real world projects can be found at SAV repository in Github: <https://github.com/sunjun-group/SAV.git> (in folder /Ziyuan/testdata).

How to use:

* By default all testdata from /etc/testdata.csv will be loaded. To load data from another configuration file call TestPackage.loadTestData(String testDataFile) , new configuration for a project will override the existing one.
* To get configuration for a project call TestPackage .getPackage(String projectName, String bugNo).

## Special Components

### JunitRunner

JunitRunner is the component used for running testcases, also support for loading in a separate process.

In order to start JunitRunner, sav.junit.runner.jar(in sav.commons/src/main/resources) must be included in the classpath.

How to build sav.junit.runner.jar:

* Package sav.junit.runner.jar is a shaded jar of sav.commons module with all its classes and necessary external classes required for junit. The sav.junit.runner.jar is under sav.commons/src/main/resources.
* To build and create this shaded jar, run sav\_shaded\_jar (should be launched automatically into your run configurations of Eclipse, it is saved under tzuyu.parent/launches).

How to start JunitRunner:

AppJavaClassPath appJavaClassPath = initAppClassPath();

JunitRunnerParameters junitParams = new JunitRunnerParameters();

junitParams.setJunitClasses(junitClassNames);

junitParams.setTestingPkgs(testingPkgs);

junitParams.setTestingClassNames(testingClassNames);

JunitResult jresult = JunitRunner.runTestcases(appJavaClassPath, junitParams);

* To create vmConfiguration to run JunitRunner from AppJavaClasspath, call sav.strategies.junit.SavJunitRunner.createVmConfig(AppJavaClassPath appClasspath).

# Dependencies

## Java Slicer

- Java Slicer is a dynamic java slicing tool.

Home page: <https://www.st.cs.uni-saarland.de/javaslicer/>

Source code: <https://github.com/hammacher/javaslicer>

- JavaSlicer is not deployed in any public maven repository, so we deployed it on to our nexus server at

<http://202.94.70.100:8081/nexus/content/repositories/thirdparty/> , and configure this thirdparty repository in our tzuyu.parent/pom.xml. In case this repository is not available, we can use maven to install the jar files into our local repository, check the script in /etc/win-mvn-install-libs.bat or /etc/linux-mvn-install-libs.sh. The detail of running these scripts is also described in the installation note.

- JavaSlicer packaged jars are in /etc/libs/javaslicer/.

- Using JavaSlicer: Firstly, we start a new jvm process to run testcases with tracer agent loaded to produce a trace file. Then, we call slicer to slice from input entry points from trace file.

Notes:

* Jdk supported version :
* As tested, javaSlicer works fine with jdk1.6, jdk7.
* For jdk 1.8 which introduces new language features, JavaSlicer cannot do instrument.
* Jdk verification during running testcases :
  + In case our sav.junit.runner.jar and the analyzing project are compiled with different version of jdk, during running testcase to create trace file, jdk can do the verification on loaded classes, this will raise a verifier error.
  + To turn off class verification, we added the jvm option –noverify (see JavaSlicerVmRunner)
  + From jdk8, the option –noverify is not supported any more, so the best way to avoid this kind of problem is providing user multiple sav.junit.runner.jar compiled by different version of jdk.

## Wala slicer

Wala slicer is a static java slicing tool.

Wiki page: <http://wala.sourceforge.net/wiki/index.php/UserGuide:Slicer>

Source code: <http://sourceforge.net/projects/wala/>

- The same issue with javaSlicer, which does not exist in a public maven repository. The packaged jars can be found in /etc/libs/wala, and the script to install is in /etc/win-mvn-install-libs.bat or /etc/linux-mvn-install-libs.sh.

- Basically, wala needs entry points which are the first line of code where the slicing begins and we need to define the scope for wala to build the class hierarchy, the scope should be the application jars (the application itself and its dependencies), jre libs, and exclude some common jdk packages to prevent the class hierarchy is built too big and unnecessary.

- The example can be found on wala’s wiki page or in our class slicer.wala.WalaSlicer in module slicer.walaslicer.

## JaCoCo

Java Code Coverage library.

Home page: <http://www.eclemma.org/jacoco/>

Source code: <https://github.com/jacoco/jacoco/tree/master>

All junit tests will be executed in a new process with jacoco agent. After the execution, a file (tzJacoco.exec) which contains analyzed information is created in the temp folder. ExecutionDataReporter then is used to analyze the execution data and report upon purpose.

## JPDA(Java Platform Debugger Architecture)

JPDA – Structure overview: <http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/architecture.html#components>

- Requires: jdk tools.jar.

In pom.xml, we add a dependency with scope system (similar to provided scope, but with defined jar path).

- The scenario using for debugging: the connection established between the front and back-ends is attaching connector (Target VM attaches to previously-running debugger).

Notes:

* JVM crashes with JDWP exit error:
  + The error is the same with the error shows up when you launch eclipse after closing it when some program still running in debug mode. You will see this in the log file: “JDWP exit error AGENT\_ERROR\_INVALID\_EVENT\_TYPE(204)”.
  + In our program, in order to limit the execution time for a task, we use sav.common.core.utils.ExecutionTimer.java which calls thread.stop() to kill thread instead of thread.interrupt() because interrupt() cannot kill infinitive loop. And jvm will crash if the thread stops right at the time it is being suspended, waiting for breakpoint handler in debugger. Jdk deprecated stop(), but they haven’t provided any helpful solution to handle infinitive loop yet! While because we need to modify variable value at runtime, it is easy to create a loop.
  + The problem does not always happen. It only occurs when the timeout of the execution of a test case is reached; the breakpoint event is caught and handling.
  + For now, to work around this problem, when the crash happens, we ignore that execution result.
* BreakpointDebugger - "Time out! Cannot get event set!"
  + When getting eventSet from eventQueue, eventQueue.remove() can run forever, so we set timeout = 3s, adjust this timeout if needed.

## JavaIlp

<http://javailp.sourceforge.net/>

Read ../etc/libs/javailp/javailp-native/readme.txt for the guide to run javailp.

You can find the examples for JavaIlp on their website above.

## Slf4j (Simple Logging Facade for Java)

<http://www.slf4j.org/>

slf4j-log4j-binding is being used in test, in order to enable log4j in test, check sav.commons. AbstractTest.init().

The configuration for log4j for test is in test-log4j.properties.

Logs will be store in [running-module]/logs/tzuyu-test.log

# Maven:

Our project is configured to build by maven in order to take advance of many of its useful build plugins and its deploying feature to public repository for other user to use.

Home page: <http://maven.apache.org/>

## Nexus server

Nexus server is a maven repository manager.

The server has been using to store project dependencies not available at any public maven repository such as javaslicer, wala slicer, and also to deploy our project for sharing to the community.

Documentation: <http://books.sonatype.com/nexus-book/reference/>

The installation guide for nexus server can be found here: <http://bneijt.nl/blog/post/nexus-on-ubuntu-12.10-installation/>

The current nexus server setting can be found in this file: /etc/maven/settings.xml.

## Deploy a library to nexus server

Requirements:

* Make sure you have maven on your machine. If not, download the binary zip file from its page.
* Maven settings ([maven-folder]/conf/settings.xml) contain the server configuration.

Example:

<servers>

<server>

<id>snapshots</id>

<username>Ziyuan</username>

<password>...</password>

</server>

<server>

<id>thirdparty</id>

<username>Ziyuan</username>

<password>...</password>

</server>

</servers>

<profiles>

<profile>

<id>nexus-sever</id>

<repositories>

<repository>

<id>snapshots</id>

<url>...</url>

</repository>

<repository>

<id>thirdparty</id>

<url>...</url>

</repository>

</repositories>

</profile>

</profiles>

<activeProfiles>

<activeProfile>nexus-sever</activeProfile>

</activeProfiles>

Notes:

* If it fails in the middle of the deployment, make sure that failed uploaded folder is deleted on the server before starting the deployment again.
* If you get error 400 "bad request" when deploy the library, these are some possible reasons:

+ user credentials are wrong

+ url to server is wrong

+ user does not have access to the deployment repository

+ user does not have access to the specific repository target

+ artifact is already deployed with that version if it is a release (not -SNAPSHOT version)

+ the repository is not suitable for deployment of the respective artifact (e.g. release repo for snapshot version, proxy repo or group instead of a hosted repository)

### Deploy a third-party library:

* Use: mvn deploy:deploy-file
* Check the example in /etc/win-mvn-deploy-libs.bat

### Deploy project to the server

Choose either one of these ways:

1. In eclipse, select Run/ Run Configuration/ Maven Build/parent deploy.

(“parent deploy” must be available since the launch “parent deploy.launch” defined under /app/tzuyu.parent/launches will be automatically imported into run configurations in eclipse)

1. Open Cmd, go to /etc folder, run win-mvn-deploy-Ziyuan.bat

## Pom.xml

### Use classes in test folder of another module

In order to inherit another module’s test class, we have to configure pom.xml for mvn to build a test-jar which also includes test classes, and add this as a dependency in our module.

<https://maven.apache.org/guides/mini/guide-attached-tests.html>

The downside of this solution is that you don't get the transitive test-scoped dependencies automatically. Meaning if you have some dependencies in module A with scope test, and our module B depends on module A test-jar, you have to add those dependencies as well otherwise we will get classNotFound exception.

For example:

we want test module of sav.commons is available for others, in sav.commons/pom.xml, we define maven-jar-plugin in build category to build sav.commons-test.jar

<build>

<plugins>

<!-- create test jar -->

<plugin>

<artifactId>maven-jar-plugin</artifactId>

</plugin>

</plugins>

</build>

And in the other module we need to declare the sav.commons-test dependency like this:

<dependencies>

...

<dependency>

<groupId>tzuyu-project</groupId>

<artifactId>sav.commons</artifactId>

<type>test-jar</type>

</dependency>

</dependencies>