Development note

Contents

[Ziyuan 1](#_Toc432423739)

[Set up 1](#_Toc432423740)

[Build and deploy 2](#_Toc432423741)

[Structure 2](#_Toc432423742)

[TestData 2](#_Toc432423743)

[Special Components 3](#_Toc432423744)

[JunitRunner 3](#_Toc432423745)

[Dependencies 4](#_Toc432423746)

[Java Slicer 4](#_Toc432423747)

[JaCoCo 4](#_Toc432423748)

[JPDA(Java Platform Debugger Architecture) 4](#_Toc432423749)

[JavaIlp 5](#_Toc432423750)

[Slf4j (Simple Logging Facade for Java) 5](#_Toc432423751)

[Code 5](#_Toc432423752)

[mvn: 6](#_Toc432423753)

[Deploy a library to nexus server 6](#_Toc432423754)

[Use classes in test folder of another module 6](#_Toc432423755)

# 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

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

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

- First of all, 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.

## 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, then we use ExecutionDataReporter 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

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

## 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

Code:

- Don't use classes in test source in java source (ex: TestConfiguration), otherwise, it will fail when building with mvn.

- Don't use org.junit.Assert in source code, if you want assertion use sav.common.core.utils.Assert instead.

- Put test classes which are just the input for a test case into testdata package. All tests in testdata package will be excluded during mvn build process.

<plugin>

<artifactId>maven-surefire-plugin</artifactId>

<version>${maven-surefire-plugin-version}</version>

<configuration>

<excludes>

<exclude>\*\*/testdata/\*\*/\*.java</exclude>

</excludes>

</configuration>

</plugin>

# mvn:

## Deploy a library to nexus server

Example :

## 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.

Mvn plugin

maven-jar-plugin: create specific jar (test-jar)

run mvn with goal jar:test-jar