ECE 322 Software Resources

Reference Document

Dia

Available for windows, mac and linux.
Download available at: http://projects.gnome.org/dia/

Description: Dia is a powerful graphing tool providing users with a clean and robust interface for creating anything from UML diagrams to flowcharts. It has an extensive library of shapes and connectors which are all easy to use as well as the ability to add additional third party shape libraries.

JUnit Information

Javadocs: http://junit.sourceforge.net/javadoc/
Getting started: https://github.com/junit-team/junit4/wiki/getting-started
JUnit “Cookbook”: http://junit.sourceforge.net/doc/cookbook/cookbook.htm
Cookbook is basically a tutorial, other tutorials are easy to find online.

The JUnit framework is installed in eclipse to allow test suites be to run as part of a default eclipse installation so there should be no extra software required. Some older versions of eclipse may require a jar to be imported and added to your project’s build path.

Creating Test Cases:

Creating JUnit test cases from an existing Eclipse project is simple; right-click on the project folder, and select JUnit Test case from the new submenu. This will create a default test case template with a single test in it which should always fail.

You can test that you have JUnit setup correctly at this point by right clicking your project, selecting run as, and then selecting JUnit test. The result should be a single test case run with a failed result. You can also change the fail condition to assertTrue(true) to ensure you are set up correctly.

Multiple test cases files can be added to a single project if you wish to divide your test cases into logical test suites for easier organization.

Setting up, tearing down, and adding test cases:

Modern JUnit test suites use java '@' annotations in the method headers to tell the JUnit framework which methods you want run how and when.

@Test: This annotation indicates that the tagged method should be run as a test case, test cases must contain some form of assertion.

@Before: This annotation indicates that the tagged method should be run before running the test cases in this suite. It can be used as a common setup method.

@After: This annotation indicates that the tagged method should be run after running the test cases in this suite. It can be used as a teardown method.

Additional annotations exist and can be found in the JUnit java docs which are linked to at the beginning of this section.

Adding Pass and Fail conditions:

JUnit provides a number of methods which allow the test writer to check the state of the test at various points in your test suite. These include a range of assertion methods as well as some additional convenience calls such as *fail()*.

The following list will provide a few options for this purpose:

assertTrue(<Expression>): This method asserts that the boolean expression passed evaluates to true. Test fails otherwise.

assertFalse(<Expression>): The opposite of assertTrue, same idea.

assertEquals(<Value>, <Value>): Asserts that the two objects passed to the method are equal. Uses the <dot>equals() method to determine this. Test fails if not equal.

assertNotNull(<Expression>): Asserts that the expressing passed does not have a value of null. AssertNull also exists to do the opposite.

fail(<Message>): Unconditionally causes the test case to fail. Useful for detecting situations that should never happen such as exceptions.

Obviously, the full, very extensive, list is available in the javadocs linked to at the beginning of this section.

<u>Mutation testing - PIT</u>

Pi Test (referred to as PIT) is available as a JAR

Available at: http://pitest.org/downloads/
Quick start: http://pitest.org/quickstart/

PIT provides a powerful but easy to use interface for mutation testing with Java applications and JUnit test cases. PIT can be used in one of two ways, either through a series of command line operations, or through an eclipse pluggin (still in alpha testing)

In addition to creating code mutations and running the projects JUnit test cases against the mutations, PIT also produces well formed HTML reports which allow developers to see the statement coverage of their test cases as well as how many mutants were killed, and were living mutants remain.

PIT can perform a large number of different types of mutation operations. The complete list, and descriptions of each mutation type, is available at http://pitest.org/quickstart/mutators/.

Eclipse pluggin:

A pluggin has been developed for PiTest by a third party, it is available on the eclipse market place: https://marketplace.eclipse.org/content/pitclipse

Maven:

See the documentation at:

http://pitest.org/quickstart/maven/

```
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>LATEST</version>
    <configuration>
        <targetClasses>
            <param>com.your.package.root.want.to.mutate*</param>
        </targetClasses>
        <targetTests>
            <param>com.your.package.root*</param>
        </targetTests>
    </configuration>
</plugin>
```

Gradle (third party)

http://gradle-pitest-plugin.solidsoft.info/

Read the results: If the test complete with no errors and everything was set up correctly, the terminal will provide a text report on the number of mutations created, killed, and not covered for a variety of categories. The results folder specified on the command line will contain a much more detailed HTML report on exactly where test coverage failed.

If you run into issues please refer to the libraries documentation which may be very helpful.

Web Testing Software

Node, Mocha, Express, and Chai all have excellent online resources:

- Node Guides: https://nodejs.org/en/docs/guides
- Express Getting Started: https://expressjs.com/en/starter/installing.html
- Express Guide: https://expressjs.com/en/guide/routing.html
- Mocha Getting Started: https://mochajs.org/#getting-started
- Mocha Assertions: https://mochajs.org/#assertions
- Chai Guide: http://chaijs.com/guide/styles/

Setting up and running a Node/Expresser server is relatively easy. Once NodeJs is installed on your machine (it should already be installed on the lab machines), download the project and unpack it into a directory.

From here we can use the node package manager (npm) to install the dependencies of the project. This is easily done by running the following command:

```
>> npm install
```

This command will fetch and install the required libraries, for the lab this includes the installation of ExpressJs and ChaiJs. If the command is giving you issues, individual modules can be installed as shown in the documentation provided above.

MochaJs is required to be installed globally (available as a system library), and as such it is already installed on the lab machines. If you are using your own machine you will need to install it globally using:

```
>> npm install mocha -g
```

Note that this command may require sudo. Once the applications dependencies are installed, the server can be run using node from the root directory of the project:

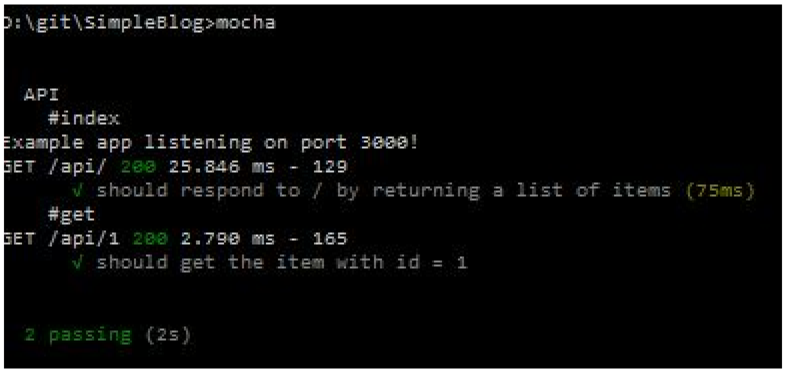
>> node app.js

This will start the http server on port 3000. A rudimentary user interface can then be accessed through your browser at http://localhost:3000.

Mocha tests are run from the projects root directory with the command:

>> mocha

A successful test execution should result in output similar to the following:



```
D:\git\SimpleBlog>mocha


  API
    #index
Example app listening on port 3000!
GET /api/ 200 25.846 ms - 129
      √ should respond to / by returning a list of items (75ms)
    #get
GET /api/1 200 2.790 ms - 165
      √ should get the item with id = 1


  2 passing (2s)
```

Notably with one entry per test case in your mocha test files. Mocha automatically searches your project tree for test cases and executes them, so the location of the test case file does not need to be specified.

Mocha has an add on library called Chai which provides a huge number of utility methods to make mocha testing easier and more readable. These functions are well outlined in the Chai documentation linked previously in this section. Some examples include:

- Checking object type: `res.body.should.be.a('array');`
- Checking http code: `res.should.have.status(200);`
- Checking for object properties: `res.body.should.have.property('id');`
- Asserting equality: `res.body.id.should.equal(1);`

These functions can be used in conjunction to intelligently and concisely assert important facets of each test.

Mocha test cases are divided into categories with descriptive subsections using the describe function:

>> `describe('API', function() {...}`

Describe blocks also allow for before and after setup functions which can be used to start and stop your node server:

```
before(function () {
    app = require('../app');
});
```

Describe blocks should be created hierarchically, with sub-blocks describing more specific server functionality being tested:

```
describe('#index', function () {...}
```

Contained within the API block, for example. Individual test cases are then specified using the "it" function:

```
it('should respond to /api/ by returning a list of items', function(done) {...}
```

And assertions made within the it callback function determine whether or not a test case passes or fails.

For more information on testing with Mocha/Chair, thoroughly review the documentation provided on each project's website.