# Testing Java with Jython and PyUnit

André Burgaud

2007-11-25

JUnit, the unit test framework written by Erich Gamma and Kent Beck, is not the only alternative for unit testing in the Java environment. This article provides a simple demonstration on how to use Jython, PyUnit and Ant to unit test a Java project.

This article assumes that the reader has some basic knowledge of unit testing, Java, Jython or Python and possibly Apache Ant. For more information about each of those technologies, see section Resources at the end of this article.

`JyUnit`, the companion code for this article, is a rewrite of the unit test example provided with JUnit, `MoneyTest`. It also includes an Ant file, `build.xml`, to facilitate the integration and automation of Jython and the unit test process.

## Requirements - Installation

To try `JyUnit`, you need:

- **Java 2 SDK, Standard Edition**:
  1. Download and install the Java Development Kit (JDK), Version 1.4.2 or greater
  2. Create an environment variable `JAVA_HOME`, pointing to the installation path of the JDK. This simplifies the installation of Apache Ant
- **Apache Ant**:
  1. Apache Ant is only needed to execute the tests from Ant and to demonstrate the possibilities of Jython integration with Ant. Without Ant, you can perform the tests using scripts *.bat* for Windows or *.sh* for UNIX/Linux.
  2. Download and install [Apache Ant]((https://ant.apache.org/)
  3. Add the directory *bin* of the `ant` directory installation to your `PATH` (For instance: `C:\ant\bin`, if Ant is installed in `C:\ant` on Windows)
- **Jython**:
  1. Download and install Jython (recommended version: Jython 2.1)
  2. Create an environment variable `JYTHON_HOME` set to the installation directory of Jython. Some JyUnit files are using this environment variable
  3. In the Jython `registry` file, located in the install directory of Jython, modify the property `python.security.respectJavaAccessibility` and set it to `false`. Read section Accessibility for more details about this change.
- **JUnit**:
  1. Download and install JUnit.
  2. If you extract the package `junit3.8.1.zip` in the directory `C:\` on Windows, you will get a new directory `C:\junit3.8.1`. JUnit is not needed to take advantage of the concept described in this article, but for demonstration purpose the unit test files included in the JyUnit package are precisely using the example provided with JUnit, `Money`. This to facilitates the comparison between the Jython solution and the Java solution to perform unit testing.
- **JyUnit** (Companion code for this article):
  1. Extract the archive jyunit02.zip or jyunit02.tgz in the JUnit installation directory in order to have the new directory `jyunit` at the same level as the directory `junit`. This does not impact any

JUnit files.
2. As an example of JUnit installed on Windows in `C:\junit3.8.1`, you get a new directory: `C:\junit3.8.1\jyunit`

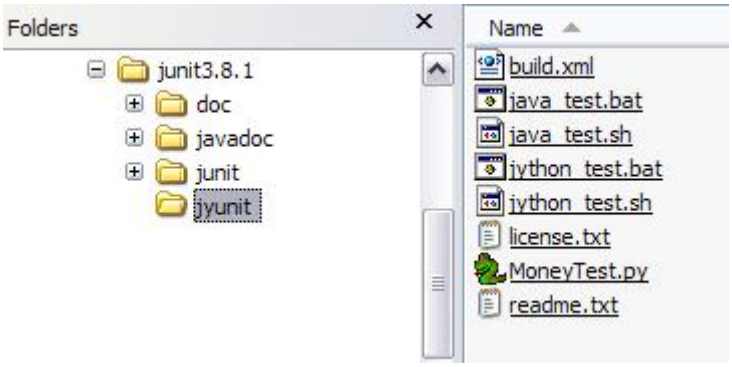


Figure 1: Directories and files architecture

## JyUnit Files Description

| File Name | Description |
|---|---|
| `build.xml` | Ant project file |
| `java_test.bat` | Windows batch to start the Java unit tests (JUnit) |
| `java_test.sh` | Shell script to start the Java unit tests (JUnit) |
| `jython_test.bat` | Windows batch to start the Jython unit tests (PyUnit) |
| `jython_test.sh` | Shell script to launch the Jython unit tests (PyUnit) |
| `license.txt` | License for JyUnit files |
| MoneyTest.py | Re-write of the file MoneyTest.java in Python (Jython) |
| `readme.txt` | Readme file |

## Java Accessibility

The Jython *registry* file is located at the root of the installation directory of Jython. Set `python.security.respectJavaAccess` to `false` in the file `registry`, allows Jython scripts to gain access to non-public fields, methods and constructors of Java objects:

```
...
# Setting this to false will allow Jython to provide access to
# non-public fields, methods, and constructors of Java objects.
#python.security.respectJavaAccessibility = true
python.security.respectJavaAccessibility = false
...
```

This setting is not recommended for production applications, but is very useful in the context of unit tests. Without this setting, for each unit-test, you get the following error:

```
Traceback (most recent call last):
File "MoneyTest.py", line 55, in setUp
self.fMB1 = MoneyBag.create(self.f12CHF, self.f7USD)
AttributeError: class 'junit.samples.money.MoneyBag' has no attribute 'create'
```

The class `junit.samples.money.MoneyBag` is not declared `public`, therefore the access is limited to classes from the same package, `junit.samples.money` in the example. This is not only limited to the test imple-

mentation in Python, this is a normal Java behavior. If the Java class `MoneyTest` did not belong to the same package as `MoneyBag`, we would obtain the following error when compiling `MoneyTest.java`:

```
MoneyTest.java:9: junit.samples.money.MoneyBag is not public in junit.samples.money;
cannot be accessed from outside package
import junit.samples.money.MoneyBag;
                         ^
```

This demonstrates one interesting fact of using Jython to test Java application while keeping the testing code fully separated to the production code. With some work it is possible to obtain a similar behavior in Java, either by using the Java Reflection API, or some additional components for JUnit such as JUnit-addons or by simply having the test part of the same Java package (most commonly used).

## Unit Tests

After preparing the environment as explained above, you should be able to perform the original JUnit test by launching `java_test.bat` or `java_test.sh`:

1. Perform the same unit test written in Jython, by using `jython_test.bat` or `jython_test.sh`. The scripts execute the tests twice: once with traces and once without trace.
2. Finally, execute the Jython test from an Ant build file.

For more information related to `Money` and `MoneyTest` used to illustrate the unit tests examples, read the article JUnit Test Infected: Programmers Love Writing Tests.

### Original Unit Tests (JUnit)

The scripts *java_test.bat* or *java_test.sh*, allow to launch the unit tests provided with JUnit:

```
C:\junit3.8.1\jyunit>java_test
.....................
Time: 0.03
OK (22 tests)
```

### Jython Unit Test (PyUnit)

Jython allows to perform unit testing without dependency with JUnit because it includes PyUnit the Python unit test framework. You can verify the presence of the module `unittest.py` located in the subdirectory `Lib` of your Jython installation directory. The scripts `jython_test.bat` or `jython_test.sh` launch the Jython unit tests:

```
C:\junit3.8.1\jyunit>jython_test
====================
Test without traces
====================
.....................
----------------------------------------------------------------------
Ran 22 tests in 0.040s
OK
================
Test with traces
================
{[12 CHF][7 USD]} *2 == {[24 CHF][14 USD]} ... ok
{[12 CHF][7 USD]} negate == {[-12 CHF][-7 USD]} ... ok
testBagNotEquals (__main__.MoneyTest) ... ok
{[12 CHF][7 USD]} + [14 CHF] == {[26 CHF][7 USD]} ... ok
{[12 CHF][7 USD]} - {[14 CHF][21 USD] == {[-2 CHF][-14 USD]} ... ok
```

```
{[12 CHF][7 USD]} + {[14 CHF][21 USD]} == {[26 CHF][28 USD]} ... ok
testIsZero (__main__.MoneyTest) ... ok
[12 CHF] + [7 USD] == {[12 CHF][7 USD]} ... ok
testMoneyBagEquals (__main__.MoneyTest) ... ok
testMoneyBagHash (__main__.MoneyTest) ... ok
testMoneyEquals (__main__.MoneyTest) ... ok
testMoneyHash (__main__.MoneyTest) ... ok
{[12 CHF][7 USD]} - [12 CHF] == [7 USD] ... ok
{[12 CHF][7 USD]} - {[12 CHF][3 USD]} == [4 USD] ... ok
[12 CHF] - {[12 CHF][3 USD]} == [-3 USD] ... ok
testPrint (__main__.MoneyTest) ... ok
[12 CHF] + [14 CHF] == [26 CHF] ... ok
[14 CHF] + {[12 CHF][7 USD]} == {[26 CHF][7 USD]} ... ok
[14 CHF] *2 == [28 CHF] ... ok
[14 CHF] negate == [-14 CHF] ... ok
[14 CHF] - [12 CHF] == [2 CHF] ... ok
testSimplify (__main__.MoneyTest) ... ok
----------------------------------------------------------------------
Ran 22 tests in 0.090s
OK
```

The Jython tests without trace display the same information as JUnit and are executed with the command :

```
set CP=.;..;C:\jython\jython.jar
java -cp %CP% org.python.util.jython MoneyTest.py
```

> ***Note***: The `CLASSPATH` with value `.;..;C:\jython\jython.jar` is just an example. It assumes that Jython is installed on Windows in the directory `C:\jython` and that the JyUnit package was extracted according to the instructions above such that `junit\samples\money` can be accessed in the `CLASSPATH` with `...` To launch the test with traces, execute the following command:

```
set CP=.;..;C:\jython\jython.jar
java -cp %CP% org.python.util.jython MoneyTest.py -v
```

In this case, each method comment or docstring is displayed. For example, for the test `testBagSubtract`:

```
def testBagSubtract(self):
  """{[12 CHF][7 USD]} - {[14 CHF][21 USD] == {[-2 CHF][-14 USD]}"""
  expected = MoneyBag.create(Money(-2, "CHF"), Money(-14, "USD"))
  self.assertEquals(expected, self.fMB1.subtract(self.fMB2))
```

PyUnit displays:

```
{[12 CHF][7 USD]} - {[14 CHF][21 USD] == {[-2 CHF][-14 USD]} ... ok
```

If there are no *docstring*, PyUnit displays the method name, its class and the result:

```
testBagNotEquals (__main__.MoneyTest) ... ok
```

The method `testBagNotEquals` is implemented in the file `MoneyTest.py` as follow:

```
def testBagNotEquals(self):
  bag = MoneyBag.create(self.f12CHF, self.f7USD)
  self.assertFalse(bag.equals(Money(12, "DEM").add(self.f7USD)))
```

In `MoneyTest.py`, in order to stay as close as possible to the reference implementation in Java (`MoneyTest.java`), the Java method comments are re-written under the form of *docstring*. Methods without comment in the Java file result in methods without *docstring* in `MoneyTest.py`.

## Ant Integration

The Ant file *build.xml* includes only the Jython unit tests. The tests are the same as the ones executed with the scripts. The `build.xml` example demonstrates two ways to easily integrate Jython in an Ant build file (without a Jython Ant task).

To launch the tests, assuming that the Ant binary files are in your `PATH`, at the command line, type:

```
C:\junit3.8.1\jyunit>ant
Buildfile: build.xml
init:
test.call:
jython.call:
[java] ......................
[java] ----------------------------------------------------------------
[java] Ran 22 tests in 0.060s
[java] OK
...
```

The tests are performed four times:

1. Without trace from the AntCall Jython task
2. With trace from the *AntCall* Jython task
3. Without trace from the MacroDef Jython task
4. With trace from the *MacroDef* Jython task

*AntCall* is an Ant tasks part of the built-in Ant tasks. The version using *MacroDef* is more elegant but the *MacroDef* feature is only available since Ant version 1.6. Read the file `build.xml` provided with the sample source code to see usage examples of *AntCall* and *MacroDef*.

## Conclusion

The Extreme Programming wave raising in the late nineties is phasing out a little. Nevertheless, some components of this development approach are well alive and evolving. In particular, unit-testing is now fully integrated in modern integrated development environments (Visual Studio, NetBeans, Eclipse...). You can find unit-testing companion libraries for most of the programming languages and some languages even include unit-testing in their syntax, D Programming Language for example. The existing libraries have also been extended to address some specific issues and environments: HTTPUnit, XMLUnit... How valuable are Jython and PyUnit to test Java application knowing the large panel of unit-tests tools already available? PyUnit used with Jython does not offer a graphical user interface (GUI). There is no specific Ant task. The Java integrated development environments are mainly basing the unit testing on JUnit. At a first glance, Jython an PyUnit seem to appear more limited than JUnit to unit-test Java application.

But, despite those apparent weaknesses, Jython and PyUnit offer some interesting benefits:

- **Simplicity**: Jython and PyUnit comply with *Keep It Simple Stupid* (KISS). Unit tests are extremely valuable, but they should remain simple in order to have the main development effort focused on the final product rather than the unit tests: does the user need the unit tests?
- **Rapid Application Development (RAD)**: At the cost of getting familiar with the Python language syntax (The goal of this article is not to start any religion war for or against a specific programming language), the simplicity, clarity of syntax and absence of compilation are some of Jython compelling points to develop and perform Java unit-tests.
- **PyUnit ships with Jython**: There is no need to install other tools but Jython.
- **Code separation**: As demonstrated in the examples above, it is easy to keep a total independence between the code to be tested and the unit-tests.
- **Easy integration**: Although not available by default in the popular Java integrated development environments, it is rather easy to call Jython from Ant and to drive unit tests. Regarding the IDE's,

projects such as *Coyote* and *JyDT* could provide satisfaction in that domain for respectively NetBeans and Eclipse.

Some points listed above are not limited to Jython and PyUnit. There are other serious candidates to unit test Java applications. Among the choices, you can combine Jython and JUnit, Groovy and JUnit, BeanShell and JUnit... Those are just some examples and each of those combinations could be the topic for a full separate article.

## Download

The code for this article is available in z Zip format jyunit02.zip or TAR archive format jyunit02.tgz.

## Legal

- Jython, JUnit et Ant are free open source projects. Check the licenses for each product:
  - The Jython License](https://www.jython.org/jython-old-sites/license.html),
  - JUnit Eclipse Public License,
  - Ant Apache License 2.0.
- Java is a trademark or registered trademark of Sun Microsystems, Inc. in the U.S. and certain other countries.