

GumpDevelopment

Gump Development

Gump development is primarily in Python, see [GumpPython](#).

Gump uses Python 2.3 or above.

Overview

Use pydoc to get a look at the classes.

In \$GUMP/python, with \$PYTHONPATH set (to pwd), run pydoc:

```
> python $PYTHONlib/pydoc.py -p 1234 gump
```

then browse the WWW site (on <http://localhost:1234>) it generates to get class documentation.

Note: Currently an instance of pydoc runs on brutus:

[Gump PyDoc](#)

See also the code documentation at **GumpCode**.

Debugging

Gump uses the standard Python 'logging' package (bundled in 2.3). Typically the command line options of *--debug* and *--verbose* turn this on. Gump code current uses a single log instance (not one per package/module).

Write to the log using `log.debug( )`

A very useful feature in exception cases is the following, the `exc_info=1` (there is no `True` in Python 2.2) logs a stack trace. The details object is often informative also.

try:

...

except Exception, details:

```
log.error('Problems problems...' + str(details), \
```

```
exc_info=1)
```

Unit Testing

Unit tests (not yet converted to the real pyunit, a knock off but similar) are run using:

```
python gump/test/pyunit.py
```

One can run a single test (or set of tests) by passing a wildcarded (filename-like not regexp) expression. e.g. **Nag* for all nag tests. This matches the method (test) name, not test suite name.

Adding Unit Tests

First, create a sub-class of [UnitTestSuite](#) (in pyunit.py) and implement `init( )`, and the `setUp( )` and/or `tearDown( )` as with any other *unit style (e.g. junit). Then create methods `testXXX( )` that either raise exceptions (if they fail) or use `self.assertXXX( )` style methods (which raise exception when assertions fail).

Second (ugly) add a segment like like this to pyunit.py, to register the new suite:

```
from gump.test.xxx import XXXTestSuite
runner.addSuite(XXXTestSuite())
```

Basically, when pyunit runs it walks through all test suites attempting to match all `testXXX( )` methods to the provided pattern (or *** for all) and when it finds them, it runs them (with `setUp( )` and `tearDown( )` run before/after). Any failure (exception) is caught and reported later.

Local Integration Testing

Note: This is closer to a unit test than an integration test, but might grow closer to the latter.

1) set or export the following:

GUMP_NO_CVS_UPDATE=true

GUMP_WORKSPACE=python\gump\test\resources\full1\mine **[Note: no trailing .xml]**

2) Edit the 'mine' (or whatever you call it) workspace (copy it from the workspace.xml in same directory):

```
<?xml version="1.0" ?>
<workspace name="Adam"
    basedir="F:\data\gump-ws"
    jardir="F:\data\gump-ws\jars"
    logdir="F:\data\gump-ws\log"
    pkgdir="F:\data\gum-ws\package"
    email="ajack@apache.org"
    mailserver="mail.try.sybase.com"
    mailinglist="ajack@apache.org"
    version="0.4">

    <property name="build.sysclasspath" value="only"/>
    <sysproperty name="build.clonevm" value="true"/>

    <profile href="profile.xml"/>

    <threads updaters="1" builders="0" />
    <nag to="ajack@apache.org" from="ajack@apache.org"/>

</workspace>
```

Note: Change the e-mail address, mailing list (bad name) and mail server to your own. Also, override nagging to oneself.

3) Run

With the above, going to Gump's root and typing gumpy ought perform a reasonable test run.

Note: Currently no aspect of the workspace is building (or even updating) but that can be worked on to improve it (w/ some creativity and/or help from infr@).

Integration Testing

Go to the **test flavour** on brutus (or your own local full Gump) and run :

```
gumpy.sh -w ../minimal-workspace.xml ant [--debug]
```

to get a quick run. Once done, do:

```
gumpy.sh -w ../gump.xml all [--debug]
```