Gump3Debugging

A no-nonsense guide to debugging Gump3

RTFM

`./gump help` is your friend. We actually have useful hints in there 🙂

Write a unit test

Large parts of gump are architected to be pretty easy to test. If you truly fail to understand what is going on, simply write some test code that details what you believe should be happening.

The great thing about these tests is that after you've found and fixed a bug, they serve as a means to prevent the error from ever being re-introduced.

Loggers

The alternative to print is to use self.log.debug(msg) in most components. If a component doesn't have access to a logger, add it to __init__:

```
def MyComponent:
  def __init__(self, log): # need to add the 'log' argument
    self.log = log # need to add this line
```

and wherever the component is created (config.py in 99% of the cases) create a logger and feed it to the component:

```
  def get_my_component(config):
    from gump.my import MyComponent
    log = get_logger(config, "my") # need to add this
    return MyComponent(log) # need to add the 'log' argument
```

the advantage of using a logger over simple 'print' statements is that you can leave them in the code and commit 'em, which means others can learn from your debug Foo 🙂

State

To figure out the "state" of a gump run at any point in the program, you'll usually want to introspect the model. The good news is that you can get to the top of the model tree (the workspace) from nearly any object in the model, eg if you have a Script element, the workspace is at

```
  script.project.module.repository.workspace
```

and from there you can introspect the entire tree. Doing that introspection pretty much involves working with python lists, dictionaries and a little bit of metaprogramming. Useful tricks include the 'dir()' command, as well as 'hasattr()', 'getattr()'. It really pays off to learn a little about python list comprehensions. For example you may wish to write code like

```
class MyScriptBuilder(AbstractPlugin):
  def __init__(self, log):
    self.log = log

  def visit_project(self, project):
    for s in [c for c in project.commands if isinstance(c, Script)]:
      self.handle_script(script)

  def handle_script(self, script):
    # AARGH! What on earth is going on here????
    if script.project.name == "bootstrap-ant":
      plist = \
        script.project.module.repository.workspace.projects.values()
      for problem in [p for p in plist if check_failure(p)]:
        self.log.debug("Project %s failed!" % script.project.name)
```

"Debug probe"

I've built and committed a little plugin called the IntrospectionPlugin (which is enabled when you pass --debug on the command line) which provides a nice place to do run the kinds of debug snippets as hinted at above. It gets run last just before gump exits, so its output is last on the console.

It has some sample code in there showing off the power of a few dir() statements along with some list comprehensions. Putting your debug code in that central place (instead of directly inside your MyScriptBuilder) is very useful if you're dealing with exceptions you don't fully grok. Simply try and recreate the problem inside the IntrospectionPlugin: it can often provide useful clues. For me, often, some weird error is caused by me making some weird typo, and simply writing a little bit of code from scratch helps isolate the problem.

Use color

If you've got a terminal that supports colored output (you don't? What age are you from??), pass gump the --color argument to make log output a lot more readable. To quickly find a debug message in the many lines of output gump generates, add a little color yourself. In the above example, you'd maybe want to

```
  from gump.util import ansicolor
  self.log.debug(
    "%sProject %s failed!%s" % \
    (ansicolor.Red, script.project.name, ansicolor.Black))
```

Do note that gump pretty much is geared for black-text-on-white-background, so if you have your terminal configured with a black background, you won't be seeing much at all 🙂

Using a Real Debugger(tm) with Gump3

Running `./gump help` shows two debug-related command line options:

```
    debug           -- run pygump in debug mode, attaching pdb
    debug-with-wing -- run pygump in debug mode, attaching the Wing IDE
```

Which have docs:

```
[lsimons@giraffe]$ ./gump help debug


      Run pygump in debug mode.

      Usage:
        ./gump debug [gump.py-args ...]

   This is not the same as executing the 'run' command with a '--debug'
   parameter. Using this command will actually start the command line
   debugger pdb to run gump in, whereas the '--debug' option customizes
   the log verbosity gump will use.

   This command otherwise accepts the same arguments as the 'run'
   command.

[lsimons@giraffe]$ ./gump help debug-with-wing

                              Gump3

      Run pygump in debug mode.

      Usage:
        ./gump debug [gump.py-args ...]

  This is not the same as executing the 'run' command with a '--debug'
  parameter. Using this command will actually start the debug connector
  for the Wing IDE and attach it to the gump process, whereas the
  '--debug' option customizes the log verbosity gump will use.

  This command otherwise accepts the same arguments as the 'run'
  command.
```

Usage is something like this:

```
./gump debug -w \
  $HOME/svn/gump/branches/Gump3/fixture/metadata/workspace.xml
```

or for wing ide:

```
export WINGHOME=path/to/wing/2
./gump debug-with-wing -w \
  $HOME/svn/gump/branches/Gump3/fixture/metadata/workspace.xml
```

If you've not worked with debuggers before, you should probably take some time to review the relevant documentation for your choice of

- PDB: http://www.ferg.org/papers/debugging_in_python.html
- Wing IDE: http://wingware.com/doc/debug/index