INF 3331: Software Engineering

Ola Skavhaug, Joakim Sundnes and Hans Petter Langtangen

Dept. of Informatics, Univ. of Oslo

&

Simula Research Laboratory

August 2011

Software engineering

Version control systems

Why?

- Can retrieve old versions of files
- Can print history of incremental changes
- Very useful for programming or writing teams
- Contains an official repository
- Programmers work on *copies* of repository files
- Conflicting modifications by different team members are detected
- Can serve as a backup tool as well
- So simple to use that there are no arguments against using version control systems!

Some git commands

- git: a modern version control system, similar to mercurial, bazaar, svn, cvs etc.
- See `http://git-scm.com, http://github.com`
- `git clone URL`: clone a (remote) repository
- `git init`: create a (local) repository
- `git commit -a`: check files into the repository
- `git rm`: remove a file
- `git mv`: move/rename a file
- `git pull`: update file tree from (remote) repository
- `git push`: push changes to central repository
- And much more, see `git help`

git example 1

```
git clone git://github.com/git/hello-world.git
cd hello-world
(edit files)
git commit -a -m ’Explain what I changed’
git format-patch origin/master
(update from central repository:)
git pull
```

git example 2

```
cd src
git init
git add .
(edit files)
git commit -a -m 'Explain what I changed'
(accidentally remove/edit file.tmp)
git checkout file.tmp
```

Tests

- How to verify that scripts work as expected
- Regression tests
- Regression tests with numerical data
- `doctest` module for doc strings with tests/examples
- Unit tests

More info

- Appendix B.4 in the course book
- `doctest`, `unittest` module documentation

Verifying scripts

How can you know that a script works?

- Create some tests, save (what you think are) the correct results
- Run the tests frequently, compare new results with the old ones
- Evaluate discrepancies
- If new and old results are equal, one believes that the script still works
- This approach is called *regression testing*

The limitation of tests

Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. -Dijkstra, 1972

Three different types of tests

- Regression testing:
  test a complete application (“problem solving”)
- Tests embedded in source code (doc string tests):
  test user functionality of a function, class or module
  (Python grabs out interactive tests from doc strings)
- Unit testing:
  test a single method/function or small pieces of code
  (emphasized in Java and extreme programming (XP))

Info: App. B.4 in the course book
doctest and unittest module documentation (Py Lib.Ref.)

Regression testing

- Create a number of tests
- Each test is run as a script
- Each such script writes some key results to a file
- This file must be compared with a previously generated ’exact’ version of the file

A suggested set-up

- Say the name of a script is `myscript`
- Say the name of a test for `myscript` is `test1`
- `test1.verify`: script for testing
- `test1.verify` runs `myscript` and directs/copies important results to `test1.v`
- Reference ('exact') output is in `test1.r`
- Compare `test1.v` with `test1.r`
- The first time `test1.verify` is run, copy `test1.v` to `test1.r` (if the results seem to be correct)

Recursive run of all tests

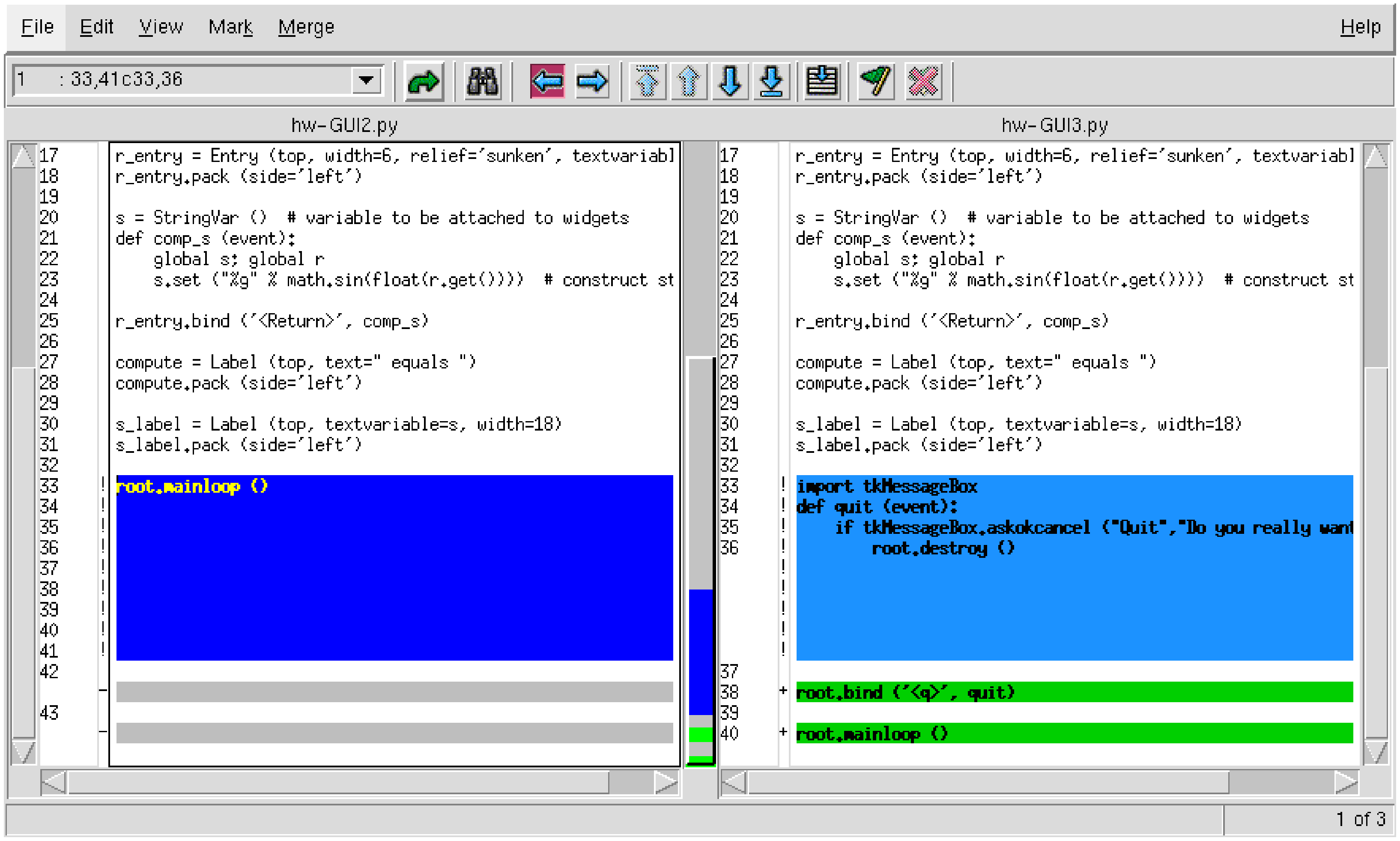
- Regression test scripts `*.verify` are distributed around in a directory tree
- Go through all files in the directory tree
- If a file has suffix `.verify`, say `test.verify`, execute `test.verify`
- Compare `test.v` with `test.r` and report differences

File comparison

- How can we determine if two (text) files are equal?
  `some_diff_program test1.v test1.r > test1.diff`
- Unix `diff`:
  output is not very easy to read/interpret,
  tied to Unix
- Perl script `diff.pl`:
  easy readable output, but very slow for large files
- Tcl/Tk script `tkdiff`:
  very readable graphical output
- `gvimdiff` (part of the Vim editor):
  highlights differences in parts of long lines
- Other tools: emacs `ediff`, `diff.py`, `windiff` (Windows only)

tkdiff

tkdiff.tcl hw-GUI2.py hw-GUI3.py



```
r_entry = Entry (top, width=6, relief='sunken', textvariabl
r_entry.pack (side='left')

s = StringVar ()  # variable to be attached to widgets
def comp_s (event):
    global s; global r
    s.set ("%g" % math.sin(float(r.get())))  # construct st

r_entry.bind ('<Return>', comp_s)

compute = Label (top, text=" equals ")
compute.pack (side='left')

s_label = Label (top, textvariable=s, width=18)
s_label.pack (side='left')

root.mainloop ()
```

```
r_entry = Entry (top, width=6, relief='sunken', textvariabl
r_entry.pack (side='left')

s = StringVar ()  # variable to be attached to widgets
def comp_s (event):
    global s; global r
    s.set ("%g" % math.sin(float(r.get())))  # construct st

r_entry.bind ('<Return>', comp_s)

compute = Label (top, text=" equals ")
compute.pack (side='left')

s_label = Label (top, textvariable=s, width=18)
s_label.pack (side='left')

import tkMessageBox
def quit (event):
    if tkMessageBox.askokcancel ("Quit","Do you really want
        root.destroy ()

root.bind ('<q>', quit)

root.mainloop ()
```

Automating regression tests

- We have made a Python module Regression for automating regression testing
- `scitools regression` is a script, using the Regression module, for executing all `*.verify` test scripts in a directory tree, run a diff on `*.v` and `*.r` files and report differences in HTML files
- Example:

```
scitools regression verify .
```

runs all regression tests in the current working directory and all subdirectories

Presentation of results of tests

- Output from the `scitools regression` command are two files:
  - `verify_log.htm`: overview of tests and no of differing lines between `.r` and `.v` files
  - `verify_log_details.htm`: detailed diff
- If all results (`verify_log.htm`) are ok, update latest results (`*.v`) to reference status (`*.r`) in a directory tree:

  ```
  scitools regression update .
  ```

- The update is important if just changes in the output format have been performed (this may cause large, insignificant differences!)

Running a single test

- One can also run `scitools regression` on a single test (instead of traversing a directory tree):

```
scitools regression verify circle.verify
scitools regression update circle.verify
```

Tools for writing test files

- Our Regression module also has a class `TestRun` for simplifying the writing of robust *.verify scripts
- Example: `mytest.verify`

```
import Regression
test = Regression.TestRun("mytest.v")
# mytest.v is the output file

# run script to be tested (myscript.py):
test.run("myscript.py", options="-g -p 1.0")
# runs myscript.py -g -p 1.0

# append file data.res to mytest.v
test.append("data.res")
```

- Many different options are implemented, see the book

Numerical round-off errors

- Consider `circle.py`, what about numerical round-off errors when the regression test is run on different hardware?

  ```
  -0.16275412    # Linux PC
  -0.16275414    # Sun machine
  ```

  The difference is not significant wrt testing whether circle.py works correctly
- Can easily get a difference between each output line in `circle.v` and `circle.r`
- How can we judge if `circle.py` is really working?
- Answer: try to ignore round-off errors when comparing `circle.v` and `circle.r`

Automatic doc string testing

- The doctest module can grab out interactive sessions from doc strings, run the sessions, and compare new output with the output from the session text
- Advantage: doc strings shows example on usage and these examples can be automatically verified at any time

Example

```
class StringFunction:
    """
    Make a string expression behave as a Python function
    of one variable.
    Examples on usage:

    >>> from StringFunction import StringFunction
    >>> f = StringFunction('sin(3*x) + log(1+x)')
    >>> p = 2.0; v = f(p)  # evaluate function
    >>> p, v
    (2.0, 0.81919679046918392)
    >>> f = StringFunction('1+t', independent_variables='t')
    >>> v = f(1.2)  # evaluate function of t=1.2
    >>> print "%.2f" % v
    2.20
    >>> f = StringFunction('sin(t)')
    >>> v = f(1.2)  # evaluate function of t=1.2
    Traceback (most recent call last):
        v = f(1.2)
    NameError: name 't' is not defined
    """
```

The magic code enabling testing

```
def _test():
    import doctest, StringFunction
    return doctest.testmod(StringFunction)

if __name__ == '__main__':
    _test()
```

Example on output (1)

```
Running StringFunction.StringFunction.__doc__
Trying: from StringFunction import StringFunction
Expecting: nothing
ok
Trying: f = StringFunction('sin(3*x) + log(1+x)')
Expecting: nothing
ok
Trying: p = 2.0; v = f(p)  # evaluate function
Expecting: nothing
ok
Trying: p, v
Expecting: (2.0, 0.81919679046918392)
ok
Trying: f = StringFunction('1+t', independent_variables='t')
Expecting: nothing
ok
Trying: v = f(1.2)  # evaluate function of t=1.2
Expecting: nothing
ok
```

Example on output (1)

```
Trying: v = f(1.2)  # evaluate function of t=1.2
Expecting:
Traceback (most recent call last):
    v = f(1.2)
NameError: name ’t’ is not defined
ok
0 of 9 examples failed in StringFunction.StringFunction.__doc__
...
Test passed.
```

Unit testing

- Aim: test all (small) pieces of code (each class method, for instance)
- Cornerstone in extreme programming (XP)
- The Unit test framework was first developed for Smalltalk and then ported to Java (JUnit)
- The Python module unittest implements a version of JUnit
- While regression tests and doc string tests verify the overall functionality of the software, unit tests verify all the small pieces
- Unit tests are particularly useful when the code is restructured or newcomers perform modifications
- Write tests first, then code (!)

Using the unit test framework

- Unit tests are implemented in classes derived from class `TestCase` in the unittest module
- Each test is a method, whose name is prefixed by `test`
- Generated and correct results are compared using methods `assert*` or `failUnless*` inherited from class `TestCase`
- Example:

```
from scitools.StringFunction import StringFunction
import unittest

class TestStringFunction(unittest.TestCase):

    def test_plain1(self):
        f = StringFunction('1+2*x')
        v = f(2)
        self.failUnlessEqual(v, 5, 'wrong value')
```

Tests with round-off errors

- Compare v with correct answer to 6 decimal places:

```
def test_plain2(self):
    f = StringFunction('sin(3*x) + log(1+x)')
    v = f(2.0)
    self.failUnlessAlmostEqual(v, 0.81919679046918392, 6,
                               'wrong value')
```

More examples

```
def test_independent_variable_t(self):
    f = StringFunction('1+t', independent_variables='t')
    v = '%.2f' % f(1.2)

    self.failUnlessEqual(v, '2.20', 'wrong value')

# check that a particular exception is raised:
def test_independent_variable_z(self):
    f = StringFunction('1+z')

    self.failUnlessRaises(NameError, f, 1.2)

def test_set_parameters(self):
    f = StringFunction('a+b*x')
    f.set_parameters('a=1; b=4')
    v = f(2)

    self.failUnlessEqual(v, 9, 'wrong value')
```

Initialization of unit tests

- Sometimes a common initialization is needed before running unit tests
- This is done in a method `setUp`:

```
class SomeTestClass(unittest.TestCase):
    ...
    def setUp(self):
        <initializations for each test go here...>
```

Run the test

- Unit tests are normally placed in a separate file
- Enable the test:

```
if __name__ == '__main__':
    unittest.main()
```

- Example on output:

```
.....
---------------------------------------------------------------
Ran 5 tests in 0.002s

OK
```

If some tests fail...

- This is how it looks like when unit tests fail:

```
===============================================================
FAIL: test_plain1 (__main__.TestStringFunction)
---------------------------------------------------------------
Traceback (most recent call last):
  File "./test_StringFunction.py", line 16, in test_plain1
    self.failUnlessEqual(v, 5, 'wrong value')
  File "/some/where/unittest.py", line 292, in failUnlessEqual
    raise self.failureException, \
AssertionError: wrong value
```

More about unittest

- The unittest module can do much more than shown here
- Multiple tests can be collected in test suites
- Look up the description of the unittest module in the Python Library Reference!
- There is an interesting scientific extension of unittest in the SciPy package