Unit Testing, Doxygen

Software Engineering and Scientific Computing

Exercises Second Day

Hanna Remmel
Institute of Computer Science
Im Neuenheimer Feld 326
69120 Heidelberg, Germany
http://se.ifi.uni-heidelberg.de
valtokari@informatik.uni-heidelberg.de

RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

software engineering heidelberg

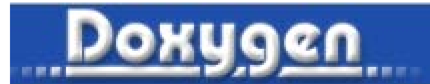
Discussion

- How was the first exercise?
  - Subversion
  - Bugzilla: creating bugs, enhancements?
  - CMake
- How was the Status-Exercise?
- Note: you can disable the email notifications in Bugzilla!

© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

Inhalt

- CppUnit
  - exists also for other languages: JUnit, Cunit, CobolUnit, JSUnit, …
- Doxygen



© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

software engineering heidelberg

CppUnit Testing: The Big Idea

- Define one method for each test
  - Method name *must* begin with "test"
  - Method must not take any parameters (other than self)
  - Shouldn't return anything
- Group related tests together in classes
  - Which must be derived from unittest.TestCase
- Call unittest.main(), which:
  - Searches the module (i.e., the file) to find all classes derived from unittest.TestCase
  - Runs methods whose names begin with "test" in an arbitrary order
    - Another reason not to make tests dependent on each other
  - Counts and reports the passes, fails, and errors

© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

Checking

- Actually check things inside test methods using methods provided by TestCase
  - Allows the framework to distinguish between test assertions, and normal assert statements
    - Since the code being tested might use the latter
- Checking methods include:
  - assert_(condition): check that something is true (note the underscore)
  - assertEqual(a, b): check that two things are equal
  - assertNotEqual(a, b): the reverse of the above
  - assertRaises(exception, func, ...args...): call func with arguments (if provided), and check that it raises the right exception
  - fail(): signal an unconditional failure

© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

Example: Checking Addition


```
import unittest
class TestAddition(unittest.TestCase):
    def test_zeroes(self):
            self.assertEqual(0 + 0, 0)
            self.assertEqual(5 + 0, 5)
            self.assertEqual(0 + 13.2, 13.2)
    def test_positive(self):
            self.assertEqual(123 + 456, 579)
            self.assertEqual(1.2e20 + 3.4e20, 3.5e20)
    def test_mixed(self):
            self.assertEqual(-19 + 20, 1)
            self.assertEqual(999 + -1, 998)
            self.assertEqual(-300.1 + -400.2, -700.3)
if __name__ == '__main__':
    unittest.main()
.F.
======================================================================
FAIL: test_positive (__main__.TestAddition)
----------------------------------------------------------------------
Traceback (most recent call last):
    File "test_addition.py", line 12, in test_positive
            self.assertEqual(1.2e20 + 3.4e20, 3.5e20)
AssertionError: 4.6e+20 != 3.5e+20
 ----------------------------------------------------------------------
Ran 3 tests in 0.000s
FAILED (failures=1)
```

Further examples in the following slides!

© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

Running Sums

- You want to test a function that calculates a running sum of the values in the list
  - Given [a, b, c, ...], it produces [a, a+b, a+b+c, ...]
- Test cases:
  - Empty list
  - Single value
  - Long list with mix of positive and negative values
- Hm...is it supposed to:
  - Return a new list?
  - Modify its argument in place and return that?
  - Modify its argument and return None?
- Your tests can only ever be as good as (your understanding of) the spec
  - Assume for now that it's supposed to return a new list

© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

Flawed Implementation


```
def running_sum(seq):
        result = seq[0:1]
        for i in range(2, len(seq)):
                result.append(result[i-1] + seq[i])
        return result
class SumTests(unittest.TestCase):
        def test_empty(self):
                self.assertEqual(running_sum([]), [])
        def test_single(self):
                self.assertEqual(running_sum([3]), [3])
        def test_double(self):
                self.assertEqual(running_sum([2, 9]), [2, 11])
        def test_long(self):
                self.assertEqual(running_sum([-3, 0, 3, -2, 5]), [-3, -3, 0, -2, 3])
F.E.
======================================================================
ERROR: test_long (__main__.SumTests)
----------------------------------------------------------------------
Traceback (most recent call last):
        File "running_sum_wrong.py", line 22, in test_long
                self.assertEqual(running_sum([-3, 0, 3, -2, 5]), [-3, -3, 0, -2, 3])
        File "running_sum_wrong.py", line 7, in running_sum
                result.append(result[i-1] + seq[i])
IndexError: list index out of range
======================================================================
FAIL: test_double (__main__.SumTests)
----------------------------------------------------------------------
Traceback (most recent call last):
        File "running_sum_wrong.py", line 19, in test_double
                self.assertEqual(running_sum([2, 9]), [2, 11])
AssertionError: [2] != [2, 11]
---------------------------------------------------------------------- R
an 4 tests in 0.001s
FAILED (failures=1, errors=1) One
```

- First implementation:
  - One failure, one error
  - Use this information to guide your diagnosis of the problem

© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

Check and Re-check

- Fix the function and rerun the tests

```
def running_sum(seq):
    result = seq[0:1]
    for i in range(1, len(seq)):
        result.append(result[i-1] + seq[i])
    return result
....
----------------------------------------------------------------
Ran 4 tests in 0.000s
OK
```

- Most first attempts to fix bugs are wrong, or introduce new bugs [McConnell 2004]
  - Continuous testing catches these mistakes while they're still fresh

© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

Eliminating Redundancy

- Setting up a fixture can often be more work than writing the test
  - The more complex the data structures, the less often you want to have to type them in
- If the test class defines a setUp method, unittest calls it before running each test
  - And if there's a tearDown method, it is run after each test
- Example: test a method that removes atoms from molecules
- Removing an atom from itself doesn't work

© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

Eliminating Redundancy: Example


```
class TestThiamine(unittest.TestCase):
     def setUp(self):
          self.fixture = Molecule(C=12, H=20, O=1, N=4, S=1)
     def test_erase_nothing(self):
          nothing = Molecule()
          self.fixture.erase(nothing)
          self.assertEqual(self.fixture['C'], 12)
          self.assertEqual(self.fixture['H'], 20)
          self.assertEqual(self.fixture['O'], 1)
          self.assertEqual(self.fixture['N'], 4)
          self.assertEqual(self.fixture['S'], 1)
     def test_erase_single(self):
          self.fixture.erase(Molecule(H=1))
          self.assertEqual(self.fixture, Molecule(C=12, H=19, O=1, N=4, S=1))
     def test_erase_self(self):
          self.fixture.erase(self.fixture)
          self.assertEqual(self.fixture, Molecule())
.E.
======================================================================
ERROR: test_erase_self (__main__.TestThiamine)
----------------------------------------------------------------------
Traceback (most recent call last):
     File "setup.py", line 49, in test_erase_self
          self.fixture.erase(self.fixture)
     File "setup.py", line 21, in erase
          for k in other.atoms:
RuntimeError: dictionary changed size during iteration
----------------------------------------------------------------------
Ran 3 tests in 0.000s
FAILED (errors=1)
```


© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

Testing Exceptions

- Testing that code fails in the right way is just as important as testing that it does the right thing
  - Otherwise, someone will do something wrong some day, and the code *won't* report it
- In Python, use TestCase.assertRaises to check that a specific function raises a specific exception
- In most languages, have to use try/except yourself
  - Run the test
  - If execution goes on past it, it didn't raise an exception at all (failure)
  - If the right exception is caught, the test passed
  - If any other exception is caught, the test failed

© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

Manual Exception Testing Example

- Example: manually test error handling in a function that finds all values in a double-ended range
  - Raises ValueError if the range is empty, or if the set of values is empty

```
class TestInRange(unittest.TestCase):
    def test_no_values(self):
        try:
            in_range([], 0.0, 1.0)
        except ValueError:
            pass
        else:
            self.fail()
    def test_bad_range(self):
        try:
            in_range([0.0], 4.0, -2.0)
        except ValueError:
            pass
        else:
            self.fail()
```


© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

Testing I/O

- Input and output often seem hard to test
  - Store a bunch of input files in a subdirectory?
  - Create temporary files when tests are run?
- The best answer is to use I/O using strings
  - Python's StringIO and cStringIO modules can read and write strings instead of files
  - Similar packages exist for C++, Java, and other languages
- This only works if the function being tested takes streams as arguments, rather than filenames
  - If the function opens and closes the file, no way for you to substitute a fake file
  - You have to design code to make it testable

© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

I/O Testing Example


```
class TestDiff(unittest.TestCase):
  def wrap_and_run(self, left, right, expected):
    left = StringIO(left)
    right = StringIO(right)
    actual = StringIO()
    diff(left, right, actual)
    self.assertEqual(actual.getvalue(), expected)
  def test_empty(self):
    self.wrap_and_run('', '', '')
  def test_lengthy_match(self):
    str = '''\
a
b
c
'''
    self.wrap_and_run(str, str, '')
  def test_single_line_mismatch(self):
    self.wrap_and_run('a\n', 'b\n', '1\n')
  def test_middle_mismatch(self):
    self.wrap_and_run('a\nb\nc\n', 'a\nx\nc\n', '2\n')
```

- Example: find lines where two files differ
  - Input: two streams (which might be open files or StringIO wrappers around strings)
  - Output: another stream (i.e., a file, or a StringIO)
- As a side effect, we've made the function itself more useful
  - People can now use it to compare strings to strings, or strings to files

© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

software engineering heidelberg

Doxygen

© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

```
//!  A test class.
/*!
  A more elaborate class description.
*/

class Test
{
  public:

    //! An enum.
    /*! More detailed enum description. */
    enum TEnum {
                 TVal1, /*!< Enum value TVal1. */
                 TVal2, /*!< Enum value TVal2. */
                 TVal3  /*!< Enum value TVal3. */
               }
         //! Enum pointer.
         /*! Details. */
         *enumPtr,
         //! Enum variable.
         /*! Details. */
         enumVar;

    //! A constructor.
    /*!
      A more elaborate description of the constructor.
    */
    Test();

    //! A destructor.
    /*!
      A more elaborate description of the destructor.
    */
   ~Test();

    //! A normal member taking two arguments and returning an integer value.
    /*!
      \param a an integer argument.
      \param s a constant character pointer.
      \return The test results
      \sa Test(), ~Test(), testMeToo() and publicVar()
    */
    int testMe(int a,const char *s);

    //! A pure virtual member.
    /*!
      \sa testMe()
      \param c1 the first argument.
      \param c2 the second argument.
    */
    virtual void testMeToo(char c1,char c2) = 0;

    //! A public variable.
    /*!
      Details.
    */
    int publicVar;

    //! A function variable.
    /*!
      Details.
    */
    int (*handler)(int a,int b);
};
```

Main Page | Classes

Class List | Class Index | Class Members

Public Types | Public Member Functions | Public Attributes

Test Class Reference

A test class. More...

List of all members.

Public Types

enum TEnum { TVal1, TVal2, TVal3 }
An enum.
More...

Public Member Functions

Test ()
A constructor.

~Test ()
A destructor.

int testMe (int a, const char *s)
A normal member taking two arguments and returning an integer value.

virtual void testMeToo (char c1, char c2)=0
A pure virtual member.

Member Function Documentation

int Test::testMe (int a,
const char * s
)

A normal member taking two arguments and returning an integer value.

Parameters:
a an integer argument.
s a constant character pointer.

Returns:
The test results

See also:
Test(), ~Test(), testMeToo() and publicVar()

virtual void Test::testMeToo (char c1,
char c2
) [pure virtual]

A pure virtual member.

See also:
testMe()

Parameters:
c1 the first argument.
c2 the second argument.

Member Data Documentation

enum Test::TEnum * Test::enumPtr

An enum.

More detailed enum description. Enum pointer.

Details.

© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

Doxygen Comment Style

- You can use /** … */ or /*! … */ or /// or //! as you like (depending on your language)
- Brief description, three possibilities
  - Use command \brief or

    ```
    /*! \brief Brief description.
    * Brief description continued.
    *
    * Detailed description starts here.
    */
    ```

  - Specia C++ style comment

    ```
    /// Brief description.
    /** Detailed description.
    */
    ```

  - If option JAVADOC_AUTOBRIEF is set, a comment block automaically starts with a brief description

    ```
    /// Brief description which ends at this dot. Details follow
    /// here.
    ```

- Documentation after members

  ```
  int var; ///< Brief description after the member
  ```

- to document global objects (functions, typedefs, enum, macros, etc), you *must* document the file in which they are defined. In other words, there *must* at least be a

  ```
  /*! \file */        OR        /** @file */
  ```


© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

Documentation Markup

\param name description	Intended for documenting function parameters. see the full sample source and documentation for how
\param name description	Intended for documenting function parameters. see the full sample source and documentation for how
\b \c \e	set the next word to bold, italic, or courier, respectively. e.g. /// You can make things \b bold, \e italic, or set them in \c courier results in You can make things **bold**, *italic*, or set them in courier.
\code \endcode	starts and ends a section of code, respectively. (it will be formatted nicely)
\n	force a newline
\internal	starts a paragraph with "internal information" (such as implementaiton details). The paragraph will be included only if the INTERNAL_DOCS option is enabled.
\mainpage	Indictaes that the following section should appear on the main page. it's a good place to introduce your most important classes, etc. (entities will be crosslinked)
\par \par Title	Starts a new paragraph (optionally with a paragraph title), works also inside other paragraphs (such as \param)

© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

Some Doxygen Options

- **EXTRACT_ALL**: if enabled, Doxygen extracts the code structure from undocumented source files.
  - for existing projects first add some documentation to the most important class declarations and methods, and then turn off.
- **JAVADOC_AUTOBRIEF**: allows to have both the brief comment and detailed description in one block (despite it's name, it works for C++ sources, too).
  - The first line of a comment block (up to the first period) is used as brief description.
- **WARN_FORMAT** set to $file($line): $text - so you can double-click doxygen warning messages in the output window to jump to the relevant source line

© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

Some special commands

- \class to document a class
- \struct to document a C-struct.
- \union to document a union.
- \enum to document an enumeration type.
- \fn to document a function.
- \var to document a variable or typedef or enum value.
- \def to document a #define.
- \typedef to document a type definition.
- \file to document a file.
- \namespace to document a namespace.
- \package to document a Java package.
- \interface to document an IDL interface.

- Usage:

```
/*! \file */    OR       /** @file */
```


© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

References

- Software caprentry (http://software-carpentry.org)
- http://www.stack.nl/~dimitri/doxygen/

© 2011 Institute of Computer Science, Ruprecht Karl University of Heidelberg

Hanna Valtokari
Institute of Computer Science
Chair of Software Engineering
Im Neuenheimer Feld 326
69120 Heidelberg, Germany
http://se.ifi.uni-heidelberg.de
valtokari@informatik.uni-heidelberg.de

RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG