Software Testing

Theory and Practicalities

Purpose

- To find bugs
- To enable and respond to change
- To understand and monitor performance
- To verify conformance with specifications
- To understand the functionality of code

Why Study Testing

- Testing is an integral part of professional software engineering
- Know how, when and why to apply testing techniques

Classifying Testing

- There are many testing techniques -- a taxonomy is useful
- White box vs. Black box
- Automated vs. Manual

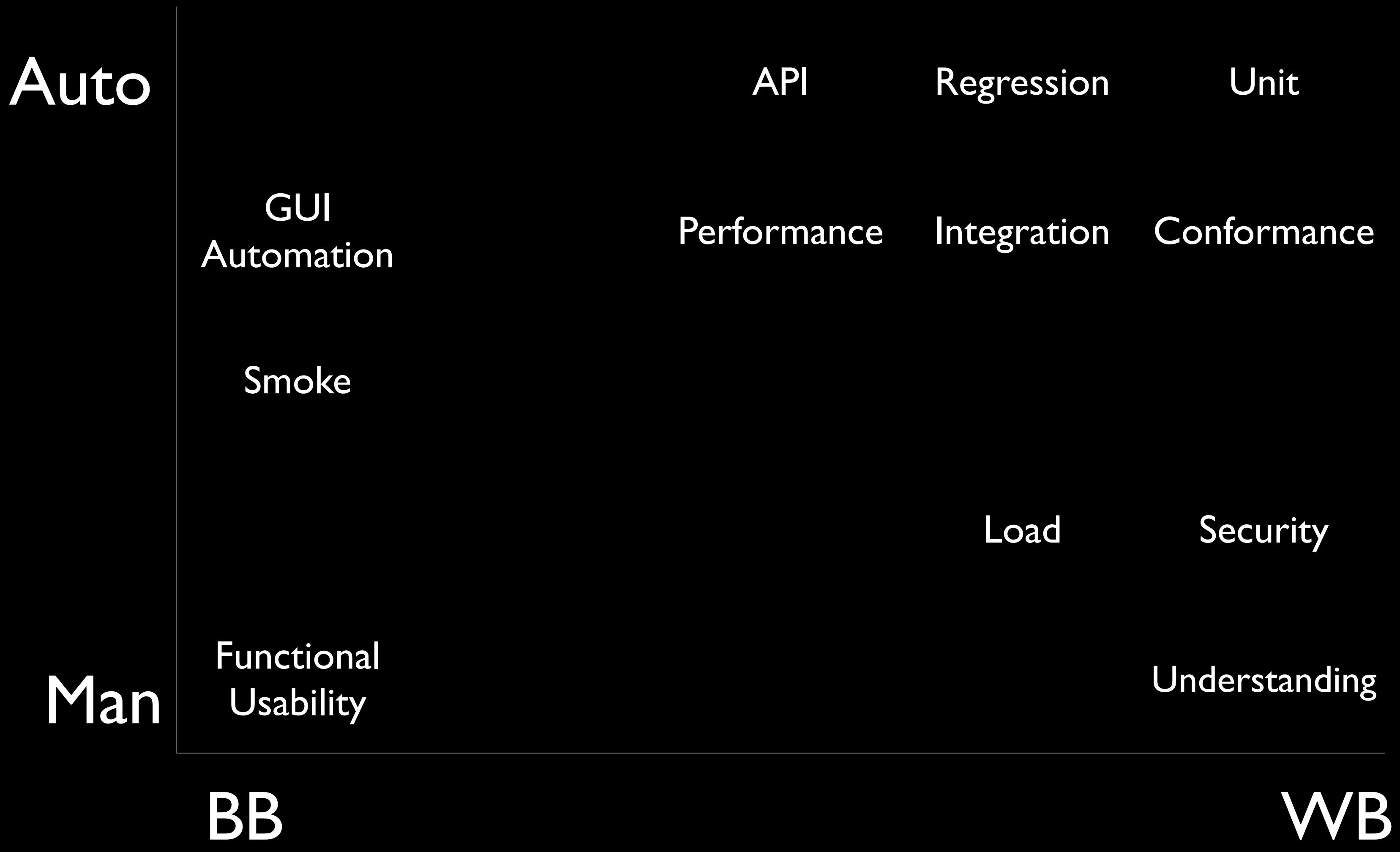
White vs. Black Box

- How deeply do you look at the implementation of the system?
- Black box: do *not want* to look at implementation
- “Grey” box - understands assumptions and limitations of system
- White box: full knowledge of code, deliberate use of implementation choices

Automated vs. Manual

- Automated: “make test”
- Manual “poke poke poke”
- Semi-automated: configure, setup, test, evaluate, retest

Test Techniques



Where to Apply These

- Roughly, the "lower" the level of the component, the more amenable to automation
- Higher level components often require significant "scaffolding" to test effectively
- Complete functional coverage of a GUI application can be very resource intensive

Unit Tests

- Ensure completeness and correctness of implementation
- Cover all public APIs and significant internal APIs
- Begin writing tests as soon as basic structure and some APIs are known
- Regression tests: bugs as a source of test

Unit Tests

- Use or create your own infrastructure to make writing unit tests trivial.
- Goal: “make test” yields true or false

Unit Tests (Example 1)

```
# ... main part of module ...

import unittest
class TestURLCanonicalization(unittest.TestCase):

    def test_basic( self ):
        u = URL( "http://www.example.com" )
        self.assertTrue( u.scheme == "http" )
        self.assertTrue( u.host == "www.example.com" )
        self.assertTrue( u.port == 80 )
        self.assertTrue( u.path == "/" )
        self.assertTrue( u.params == "" )
        self.assertTrue( u.fragment == "" )
        self.assertTrue( u.url == "http://www.example.com/" )

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

Unit Tests (Example 2)

```
# ... main part of module ...

import unittest
class TestURLCanonicalization(unittest.TestCase):

    def test_nonascii_escaped_fragment( self ):
        u = URL( "http://www.facebook.com/?ref=home#!/pages/โครงการหาบ้านใหม่ให้สุนัข
จรจัด-มก/117976974964923?sk=wall" )
        self.assertTrue( u.scheme == "http" )
        self.assertTrue( u.host == "www.facebook.com" )
        self.assertTrue( u.port == 80 )
        self.assertTrue( u.path == "/" )
        self.assertTrue( u.params == "ref=home&_escaped_fragment_=/pages/..." )
        self.assertTrue( u.fragment == "" )
        self.assertTrue( u.url == "http://www.facebook.com/?
ref=home&_escaped_fragment_=/pages/%E0%B9%82%E0...%81/117976974964923?sk=wall" )

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

API Testing

- A variant of Unit testing
- An important component of module documentation
- Added emphasis on edge-cases, exceptional conditions, parameter verification, abuse
- Well defined APIs work well with a test-first strategy

API Test Example

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    If the result is small enough to fit in an int, return an int.
    Else return a long.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]"""
   ...
    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
   ...

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Complex Integration and Unit Tests

- Test core application logic
- Often requires setting up test environment
- Invest in stubs, mocks, and scripted test scaffolding to automate complex testing

Stubs, Mocks, and Scaffolding

- Stubs: downstream fn that gives a reasonable response (null, True, ...)
- Mock: somewhat intelligent stub. e.g. Must call fnX before fnY.
- Scaffolding: potentially complex test environment. VMs, DBs, configurations

Stubs and Mocks

- More of a development than testing tool
- Focus development on one component at a time
- Enable unit and integration tests to be written

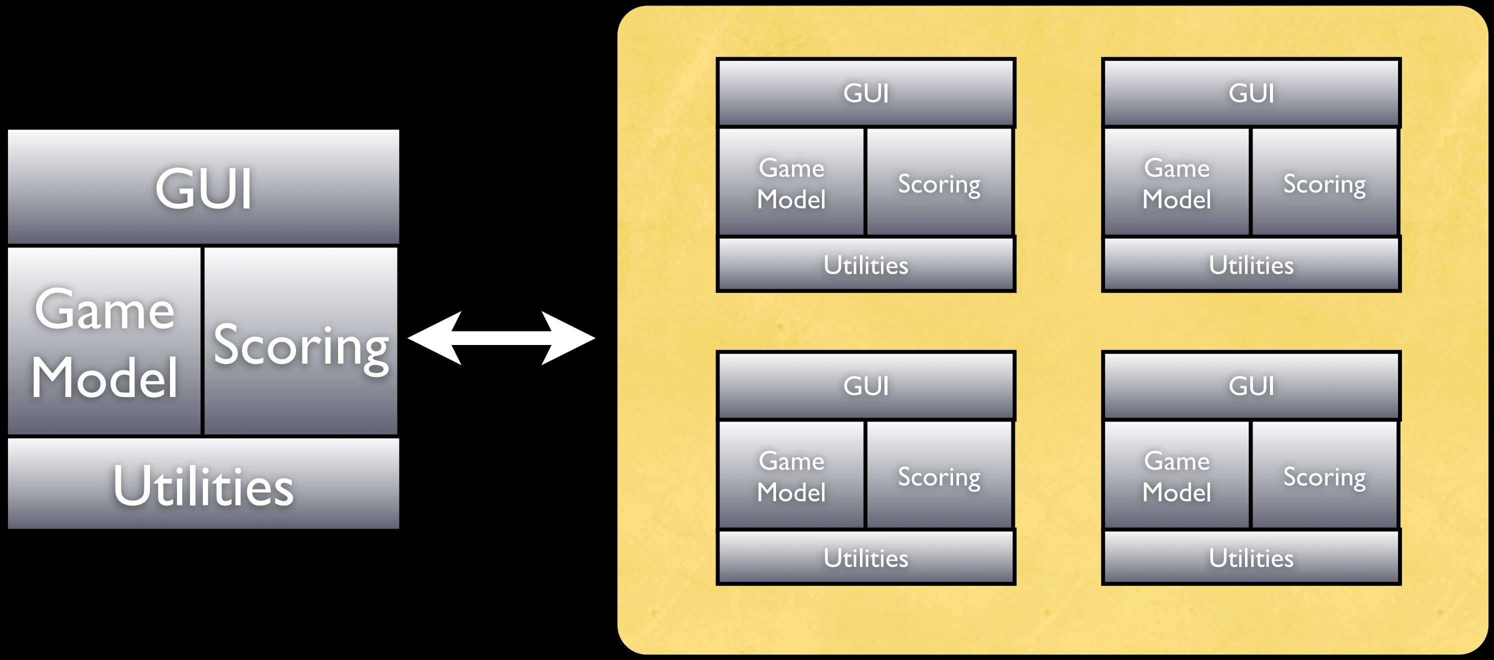
Example Stub

```
class DummyCache( object ):

    def lookup( self, key, default_value = None ):
        return default_value

    def remove( self, key ):
        pass

    def store( self, key, val ):
        pass

    def clear( self ):
        pass
```

Test Scaffolding

- Goal: a controlled version of an approximation of some “real world”
- Automate or die
- Scripting, tooling, system administration skills are required

Complex example



Scaffolding Env Example

```
Vagrant::Config.run do |config|
  config.vm.box = "precise64"
  config.vm.provision :chef_solo do |chef|
      chef.cookbooks_path = "cookbooks"
      chef.add_recipe "mysql"
      chef.add_recipe "mysql::server"
  end
  config.vm.provision :shell, :path => "./setup.sh"
end
```

```
#!/bin/sh
# setup test DB
mysql -u root -p password -e “"create database test;GRANT ALL PRIVILEGES
ON test.* TO user@localhost IDENTIFIED BY 'password';FLUSH PRIVILEGES;“
# fetch test data
curl http://example.com/test_data.csv > /tmp/test_data.csv
# load test data
mysqlimport -u root -p password --local test /tmp/test_data.csv
```

```
$ make test
# cd tests/vagrant
# vagrant up
# cd ../tests/integration
# python test.py ...
```

High-Level GUI Automation Testing

- “Robot” user - automate the use of the application
- Understand what you are testing
- Can be useful, but may be brittle

Selenium Example

```
from selenium import webdriver
from selenium.common.exceptions import TimeoutException
from selenium.webdriver.support.ui import WebDriverWait

driver = webdriver.Firefox()
driver.get("http://www.google.com")
inputElement = driver.find_element_by_name("q")
inputElement.send_keys("Cheese!")
inputElement.submit()

try:
    # wait for the page to refresh
    WebDriverWait(driver, 10).until(lambda driver :
driver.title.lower().startswith("cheese!"))

   # make assertions about the context of the page here...
finally:
    driver.quit()
```

Functional Testing

- Humans (finally!)
- Written scripts and checklists
- Levels of detail: smoke, iteration, alpha, beta, release acceptance

Often Ignored

- Logs
- Internationalization
- Stubbed exception handling

Summary

- Some testing is the developer's responsibility
- Automation makes powerful testing relatively simple