CptS 322- Software Engineering Principles I

System Testing

Instructor: Sakire Arslan Ay

Fall 2023

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Recall: Kinds of testing

- **unit testing**: looks for errors in methods, objects, or subsystems
- **integration testing**: find errors when connecting subsystems
- **system testing**: test entire system behavior as a whole, with respect to user stories and requirements
  - functional testing: test whether system meets requirements
  - performance testing: nonfunctional requirements, design goals
  - acceptance / installation testing: done by client

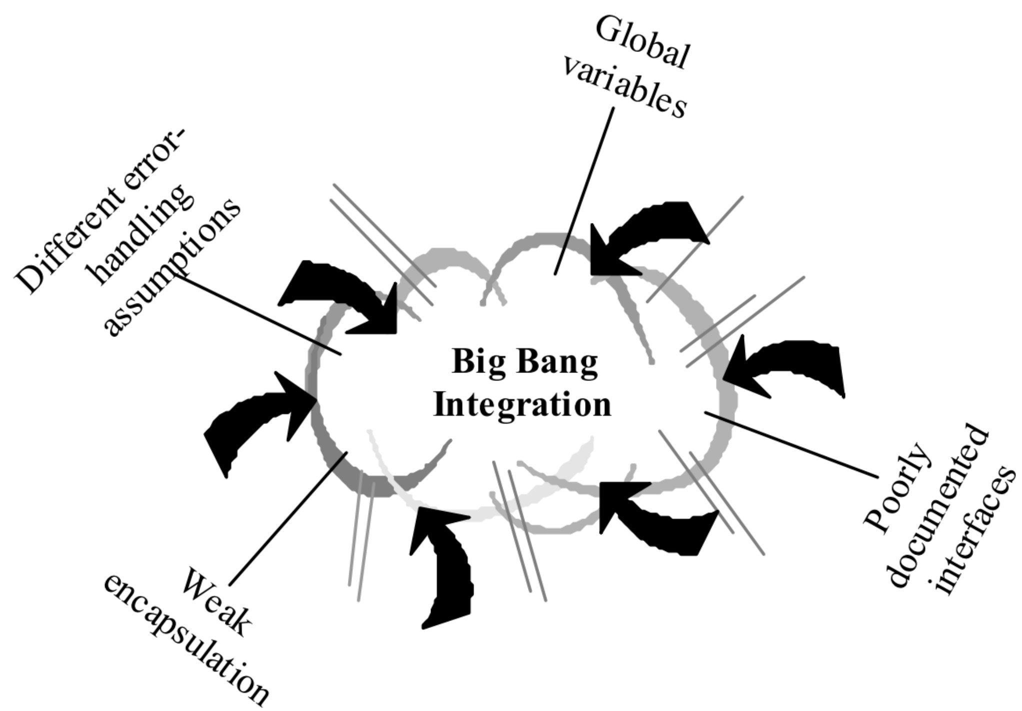
Integration

- **integration**: Combining 2 or more software units
  - often a subset of the overall project

- Why do software engineers care about integration?
  - new problems will inevitably surface
    - many systems now together that have never been before
  - hard to diagnose, debug, fix
  - cascade of interdependencies
    - cannot find and solve problems one-at-a-time

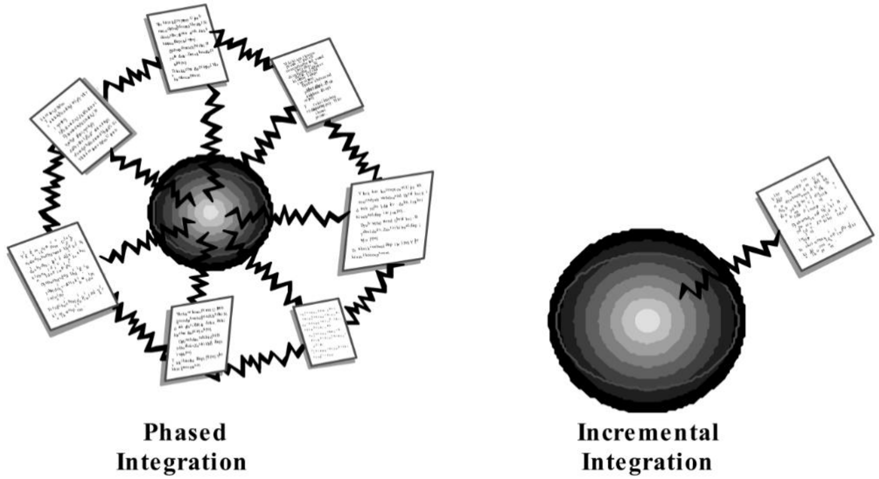
Phased Integration

- **phased ("big-bang") integration**:
  - design, code, test, debug each class/unit/subsystem separately
  - combine them all
  - and pray



Incremental Integration

- **incremental integration**:
  - – develop a functional "skeleton" system
  - – design, code, test, debug a small new piece
  - – integrate this piece with the skeleton
    - test/debug it before adding any other pieces



Benefits of Incremental Integration

- Benefits:
  - Errors easier to isolate, find, fix
  - Reduces developer bug-fixing load
  - System is always in a (relatively) working state
    - Good for customer relations, developer morale
- Drawbacks:
  - May need to create "stub" versions of some subsystems that have not yet been integrated

Daily Builds

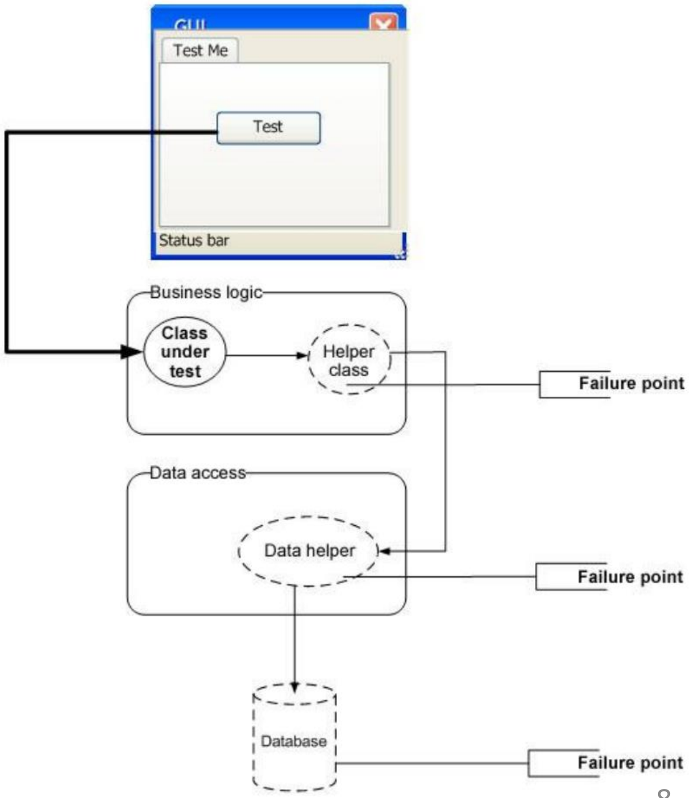
- daily build: Compile working executable on a daily basis
  - allows you to test the quality of your integration so far
  - helps morale; product "works every day"; visible progress
  - best done automated or through an easy script
  - quickly catches/exposes any bug that breaks the build

- smoke test: A quick set of tests run on the daily build.
  - NOT exhaustive; just sees whether code "smokes" (breaks)
  - used (along with compilation) to make sure daily build runs

Integration Testing

Integration testing: Verifying software quality by testing two or more dependent software modules as a group

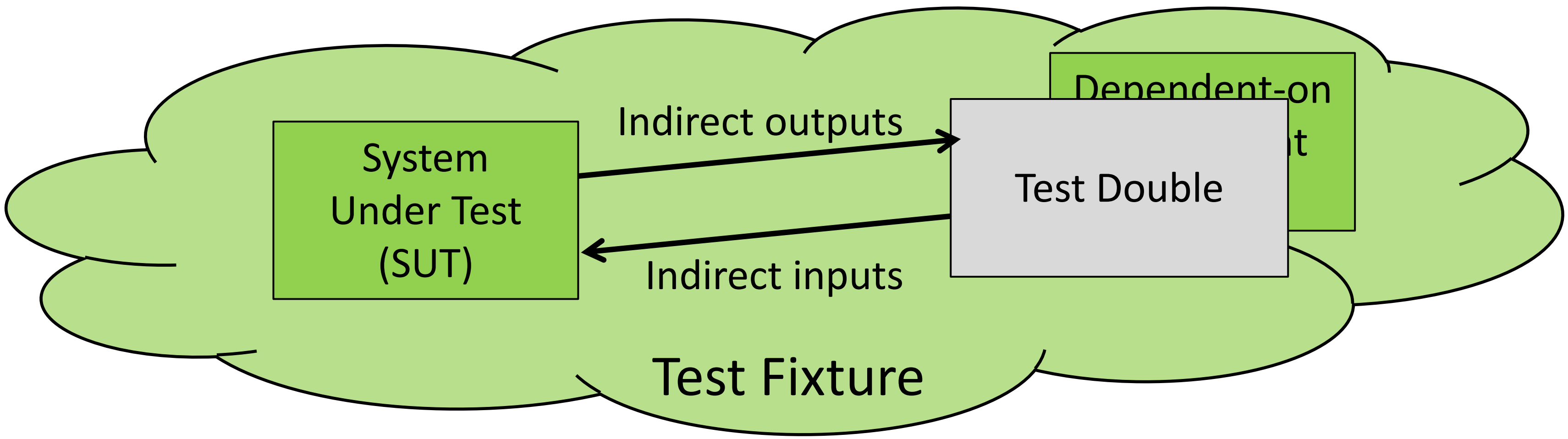
- Challenges (same as in unit testing):
  - Combined units can fail in more places and in more complicated ways.
  - How to test a partial system where not all parts exist?



How to test a partial system?

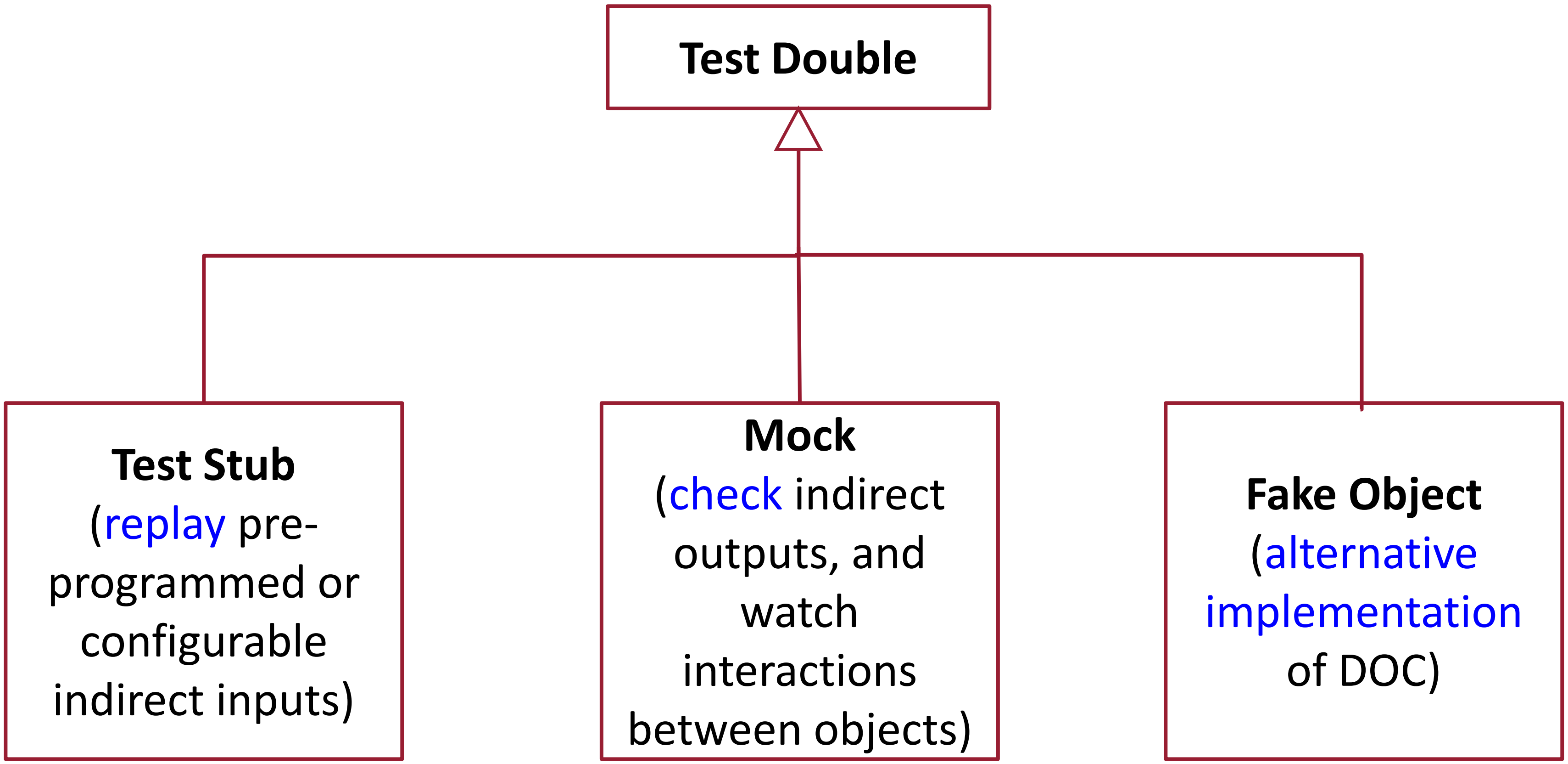
- **Use test doubles**
  - Stub: A controllable replacement for an existing software unit to which your system under test (SUT) has a dependency.
    - replay pre-programmed or configurable indirect inputs
  - Mock: A fake object that decides whether a unit test has passed or failed by watching interactions between objects.
  - Fake Object: A test double that implements (some of) the functionality of the dependent on component (DOC)
    - Alternative implementation of the DOC

Test doubles



- Double: (def.) “highly trained replacement, vaguely resembling the actor”
  - Does not need to be a very good actor (e.g., no talk)
  - Different scenarios require different degrees of conformance (skill and resemblance)

Kinds of Test Doubles



- The doubles can be hard-coded for one test, or configurable

Example application: Bugaton

- The example we will cover:
  - We want to develop a program that scans a GIT repository and tabulates the bugs that were fixed and how many lines of code were changed for each fix
    - This system is called “bugaton”, developed by George Necula (from UC Berkeley)
  - Original code uses the public “python git repository”

Bugaton

- The desired output:

```
shell> python bugaton.py
7673 182 lines changed
8202 7 lines changed
9125 18 lines changed
…
```

- The raw data is taken from "`git log --shortstat`"

```
commit db4cf7512ad65fc57a8d5685eaaee03192bb0ac2
Author: victor.stinner <victor.stinner@6015fed2-1504-0410-9fe1-9d1591>
Date: Sat Jul 3 13:36:19 2010 +0000
Issue #7673: Fix security vulnerability (CVE-2010-2089) in the audio
3 files changed, 108 insertions(+), 74 deletions(-)
```

Bugaton:

- Creating a DOC for Git repository.

```
# Our DOC
class Git:
   def cmd(self, args):
   … run "git " + args …
   … return output …
```

```
def bugaton(docGit):  # Our SUT
   log = getGitLog(docGit)
   messages = splitLog(log)
   return parseMessages(messages)

def getGitLog(docGit):
   log = docGit.cmd("log --shortstat")
   return log

def splitLog(log):
   … split log …

def parseMessages(m):
   … parse messages …
```

Example: Hard-Coded Test Stub

- Test stub for the “Git” class (DOC).

```
# Our DOC
class Git:
   def cmd(self, args):
   … run "git " + args …
   … return output …
```

```
def bugaton(docGit):
   log = getGitLog(docGit)
   messages = splitLog(log)
   return parseMessages(messages)

def getGitLog(docGit):
   log = docGit.cmd("log --shortstat")
   return log

def splitLog(log):
   … split log …

def parseMessages(m):
   … parse messages …
```

```
# Our stub for this test
class GitStub():
   def cmd(self, args):
      return "… log value…"

# Our test
def test_hard_coded_stub():
   # setup stub
   stub = GitStub()
   # exercise the SUT
   out = bugaton(stub)
   # verify
   assertEqual(out, …)
```

Example: Configurable Test Stub

```
# Configurable stub
class GitConfigStub:
   self.reply = None
   # DOC interface methods
   def cmd(self, args):
      return self.reply

   # Configuration methods
   def setReply(self, val):
      # Remember the reply
      self.reply = val
```

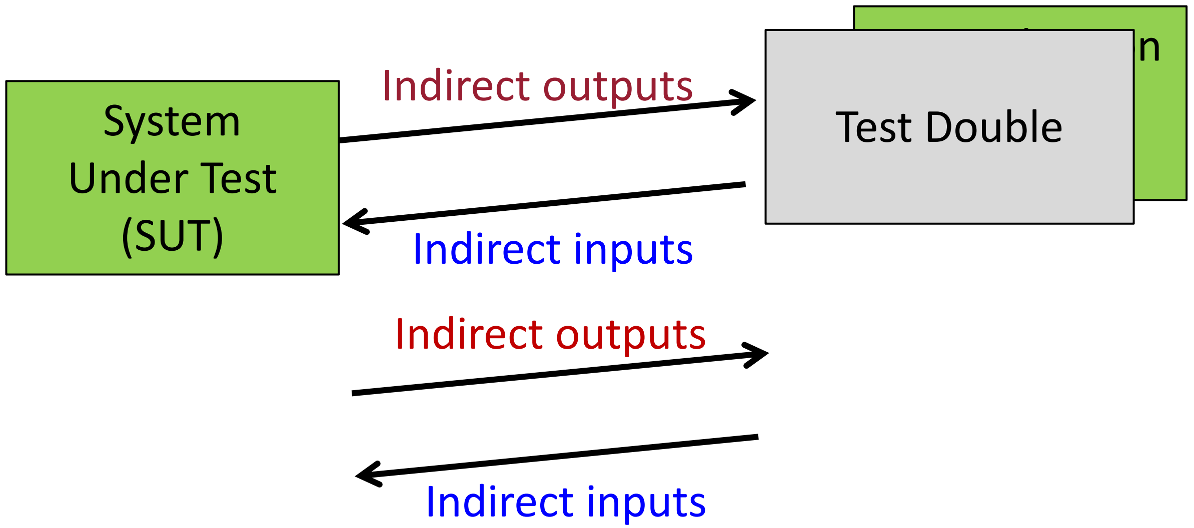
```
# Our test
def test_config_stub():
   # create stub
   stub = GitConfigStub()
   # configure stub
   stub.setReply("…log value…")
   # exercise the SUT
   out = bugaton(stub)
   # verify
   assertEquals(out, …)
```

State-Based vs. Behavior-Based Testing

- State-Based Testing (so far):
  - Setup SUT + DOC (or test double)
    - Put the SUT into a certain initial state
  - Exercise SUT
  - Check final state of SUT (and DOC)
    - Compare SUT state with expected

- Behavior-Based Testing (next)
  - A more powerful form of testing;
    - checks also the indirect interactions with the DOC
  - E.g., the order and arguments of indirect outputs

Mocks



Test Fixture

Test Mocks

- Mock Object:
  - A double that acts as an observation point for the indirect outputs
  - Monitors how SUT calls DOC
    - The sequence of calls (checks the ordering, or just the count)
    - The arguments (check their order, and values)
  - Does assertions on indirect outputs on behalf of the test
  - Checks how the SUT behaves dynamically

Mocking Frameworks

- Major choices for mocking frameworks:
  - Record-replay interface for indirect outputs
  - Fluent domain-specific language for setting expectations on indirect outputs
- One of the most notable developments in testing in the last decade
- Several frameworks for each language
  - JMock and EasyMock (for Java)
  - unittest.mock, pMock and Mox (for Python)
  - Moq and Rhino Mock (for C#)
  - etc.

Example: Testing Using unittest.mock

```
# create a mock for Git class.
# Test getGitLog function and Git.cmd method.
def test_mock_1(self):
   # create mock  for Git
   mgit = unittest.mock.Mock()
   # set return values on methods of the mock object
   mgit.cmd.return_value = "…log value…"

   # Exercise test ;  in an actual test, this will be replaced by a
   # call to SUT which calls the cmd method of Mock object "mgit".
   log = mgit.cmd("log --shortstat")

   # Verify test output
   self.assertEqual(log, "…log value…")
   #Assert that the mock was called exactly once.
   mgit.cmd.assert_called_once()
```

Example: Testing Using unittest.mock

```
#create a mock for Git class.Test its cmd method with different inputs.
   def test_mock_2(self):
      # create mock  for Git
      mgit = unittest.mock.Mock()
      # define input,expected-output pairs as a dictionary
      return_values = {"log -p rev1": "…log value1…",
                       "log -p rev2": "…log value2…",
                       "log -p rev3": "…log value3…"}

      # set return values on methods of the mock object
      mgit.cmd.side_effect = return_values.get

      # Exercise test ;  in an actual test, this will be replaced by a call
      #to SUT which calls the cmd method of Mock object "mgit".
      log1 = mgit.cmd("log -p rev1")
      log2 = mgit.cmd("log -p rev2")
      log3 = mgit.cmd("log -p rev3")

      # Verify test output
      self.assertEqual(log1, "…log value1…")
      self.assertEqual(log2, "…log value2…")
      self.assertEqual(log3, "…log value3…")
```

Example: Testing Using unittest.mock

```
def test_mock_3(self):
    # create mock  for Git
    mgit = unittest.mock.Mock()
    # define input,expected-output pairs as a dictionary
    return_values = {"log --shortstat": "…log value…",
                     "log -p rev1": "…log value1…",
                     "log -p rev2": "…log value2…",
                     "log -p rev3": "…log value3…"}
    # set return values on methods of the mock object
    mgit.cmd.side_effect = return_values.get

    # Exercise test : call the methods we expect the SUT to call
    # in an actual test, this will be replaced by a call to SUT
    log0 = mgit.cmd("log --shortstat")     # 1st call
    log1 = mgit.cmd("log -p rev1") # 2nd call
    log2 = mgit.cmd("log -p rev2") # 3rd call
    log3 = mgit.cmd("log -p rev3") # 4th call

    # Verify test output
    self.assertEqual(log0, "…log value…")
    self.assertEqual(log1, "…log value1…")
    self.assertEqual(log2, "…log value2…")
    self.assertEqual(log3, "…log value3…")

    CONTINUES ON THE NEXT SLIDE
```

Example: Testing Using unittest.mock

```
… test_mock_3 method continues…

     # verify the method calls for 'cmd'
     expected_arg_list = [
           unittest.mock.call("log --shortstat",),
           unittest.mock.call("log -p rev1",),
           unittest.mock.call("log -p rev2",),
           unittest.mock.call("log -p rev3",) ]

     # verify the number of calls
     self.assertEqual(len(mgit.cmd.call_args_list), 4)

     # verify that the 'cmd' method is called with the above 4 inputs
     mgit.cmd.assert_has_calls(expected_arg_list)

     # verify that the 'cmd' method was called  4 times
     # in order with expected inputs
     self.assertListEqual(mgit.cmd.call_args_list, expected_arg_list)
```

Mock Errors

- Unexpected method call
  - In `test_mock_2:`
    - If the SUT calls git with `mgit.cmd("log")`

      instead of `mgit.cmd("log -p rev1")`

```
AssertionError: None != '…log value1…'
```

- Missing method call
  - In `test_mock_3:`
    - If the SUT forgets to call git with `mgit.cmd(" log --shortstat")`

```
AssertionError: Calls not found.
Expected: call('log --shortstat')
```

Mock Errors

- Method calls are out of order
  - In `test_mock_3:`
- If the SUT calls git with the following (assume SUT expects a call for **rev1** before **rev2**.

```
log0 = mgit.cmd("log --shortstat")  # 1st call
log1 = mgit.cmd("log -p rev2") # 2nd call
log2 = mgit.cmd("log -p rev1") # 3rd call
log3 = mgit.cmd("log -p rev3") # 4th call
```

  - `AssertionError: Calls not found.`

Functional Testing

- **ad-hoc**: Just run the product and click things.

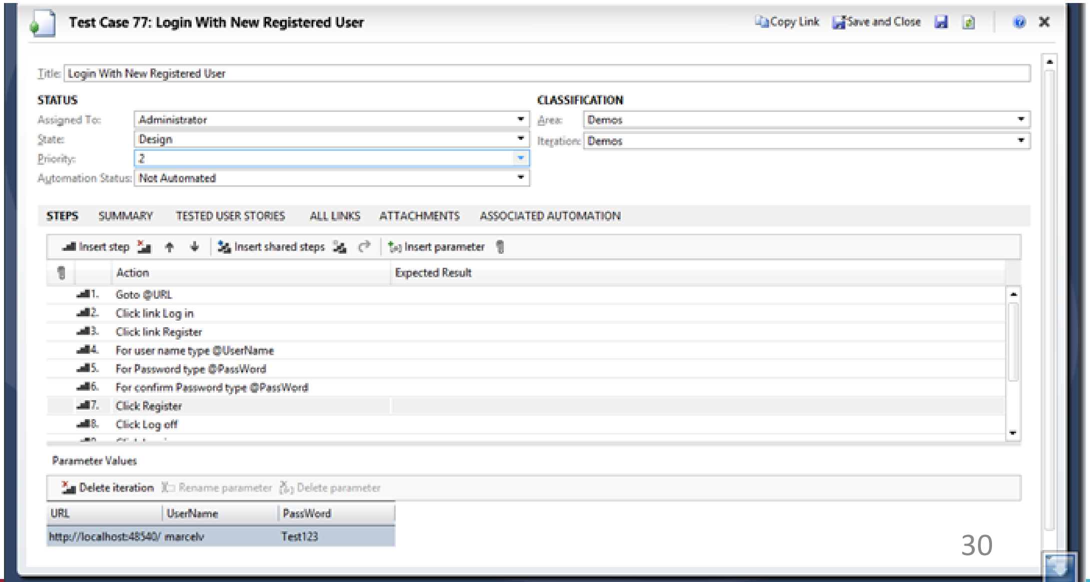
- **UI automation**: Simulate usage of a product's UI in code.
  - "record" usage and play back later
  - or write code to simulate mouse clicks

- Many developers rely too much on ad-hoc testing.
  - **pro**: Simple; fast; does not require specialized knowledge
  - **con**: Inaccurate; must be repeated many times; poor at catching regressions; costs more and more time later in the project
  - The ideal is a mix of both kinds of UI testing.

Selenium

- Records and plays back automated "test cases“ of walking through a web app's UI
- can **assert** various aspects of the web page state to make sure the page looks right

- tests can be saved as HTML
  - or can be written in:
    - Java
    - Ruby
    - Python
    - ...



Selenium Test Example (1)

```
# To install the Python client library:
# pip install -U selenium

# Import the Selenium 2 namespace (aka "webdriver")
from selenium import webdriver

# iPhone
driver = webdriver.Remote(browser_name="iphone",
   command_executor='http://172.24.101.36:3001/hub')
# Android
driver = webdriver.Remote(browser_name="android",
   command_executor='http://127.0.0.1:8080/hub')
# Google Chrome
driver = webdriver.Chrome()
# Firefox
driver = webdriver.Firefox()
```

Selenium Test Example (1) – cont.

```
# The actual test scenario: Test the codepad.org code execution service.

# Go to codepad.org
driver.get('http://codepad.org')

# Select the Python language option
python_link = driver.find_elements_by_xpath("//input[@name='lang' and
   @value='Python']")[0]
python_link.click()

# Enter some text!
text_area = driver.find_element_by_id('textarea')
text_area.send_keys("print ('Hello World!')")

# Submit the form!
submit_button = driver.find_element_by_name('submit')
submit_button.click()

# Make this an actual test.
assert "Hello World!" in driver.get_page_source()

# Close the browser!
driver.quit()
```

Selenium Test Example (2)

- Testing the Smile App UI using Selenium and PyTest
  - https://github.com/WSU-CptS-322-Fall-2023/SmileApp/blob/main/tests/test_selenium.py

- Install pytest and selenium
  - `pip install pytest`
  - `pip install selenium`

- Download the Chrome Webdriver and set the path of the webdriver directory in environment variables.
  - Make sure to download the driver compatible with your Chrome browser version
  - https://chromedriver.chromium.org/download

- Run the flask application and run the tests:
  - `python smile.py`
  - `pytest tests/test_selenium.py`

Acceptance Testing

- Acceptance testing: System is shown to the user/client/customer to make sure that it meets their needs.
  - A form of black-box system testing.
- Performance is a major aspect of program acceptance by users.

Performance Testing

- Performance is a major aspect of program acceptance by users.
  - Your intuition about what's slow is often wrong.

What's wrong with this?

```
public class Fibonacci {

    public static void main(String[] args) {
        // print the first 100,000 Fibonacci numbers
        for (int i = 1; i <= 100000; i++) {
            System.out.println(fib(i));
        }
    }

    // pre: n >= 1
    public static long fib(int n) {
        if (n <= 2) {
            return 1;
        } else {
            return fib(n - 2) + fib(n - 1);
        }
    }
}
```

Thinking about performance

- The app is only too slow if it doesn't meet your project's stated performance requirements.
  - If it meets them, DON'T optimize it!

- Which is more important, fast code or correct code?

- What are reasonable performance requirements?
  - What are the user's expectations? How slow is "acceptable" for this portion of the application?
  - How long do users wait for a web page to load?
  - Some tasks (admin updates database) can take longer

Optimization myths

- **Myth:** You should optimize your code as you write it.
  - No; makes code ugly, possibly incorrect, and not always faster.
  - Optimize later, only as needed.
- **Myth:** Having a fast program is as important as a correct one.
  - If it doesn't work, it doesn't matter how fast it's running!
- **Myth:** Certain operations are inherently faster than others.
  - `x << 1` is faster to compute than `x * 2` ?
  - This depends on many factors, such as language used.
    Don't write ugly code on the assumption that it will be faster.
- **Myth:** A program with fewer lines of code is faster.

Optimization Metrics

- **runtime / CPU usage**
  - what lines of code the program is spending the most time in
  - what call/invocation paths were used to get to these lines
    - naturally represented as tree structures
- **memory usage**
  - what kinds of objects are on the heap
  - who is pointing to them now
  - "memory leaks“
- **web page load times, requests/minute, etc.**

Benchmarking, optimization

- **benchmarking**: Measuring the absolute performance of your app on a particular platform (coarse-grained measurement).

- **optimization**: Refactoring and enhancing to speed up code.
  - I/O routines
    - accessing the console (print statements)
    - files, network access, database queries
    - `exec()` / system calls
  - Lazy evaluation saves you from computing/loading
    - don't read / compute things until you need them
  - Hashing, caching save you from reloading resources
    - combine multiple database queries into one query
    - save I/O / query results in memory for later

Avoiding Computations

- Stop computing when you know the answer:

```
found = false;
for (i = 0; i < reallyBigNumber; i++) {
    if (inputs[i].isTheOneIWant()) {
        found = true;
        break;
    }
}
```

- Hoist expensive loop-invariant code outside the loop:

```
double taxThreshold = reallySlowTaxFunction();
for (i = 0; i < reallyBigNumber; i++) {
    accounts[i].applyTax(taxThreshold);
}
```

Dynamic programming

```
public static boolean isPrime(int n) {
    double sqrt = Math.sqrt(n);
    for (int i = 2; i <= sqrt; i++)
        if (n % i == 0) { return false; }
    return true;
}
```

- **dynamic programming**: Caching previous results.

```
private static Map<Integer, Boolean> PRIME
  = ...;
public static boolean isPrime2(int n) {
    if (!PRIME.containsKey(n))
        PRIME.put(n, isPrime(n));
    return PRIME.get(n);
}
```