Using Test Doubles

CS169

Good references:
• G. Meszaros, “xUNIT Test Patterns”

Recall: Test Fixture = SUT + DOC

• What constitutes the SUT/DOC depends on the test

<table>
<thead>
<tr>
<th>Test kind</th>
<th>SUT</th>
<th>DOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit test</td>
<td>A method</td>
<td>Class constructor/other helper methods</td>
</tr>
<tr>
<td>Component test</td>
<td>Several methods/class constructor/other helper methods</td>
<td></td>
</tr>
<tr>
<td>Integration test</td>
<td>Several classes/subsystems, e.g., Google Auth/Drive</td>
<td></td>
</tr>
</tbody>
</table>

DOC Issues

• The DOC is extra “stuff” in the test
  – Need to setUp, tearDown, maintain, etc.

• The DOC might be expensive to set up
  – If you share DOCs between tests => test interference
  – E.g., database (needs to be recreated for each test)

• A bug in the DOC can lead to test failure
  – Tests are harder to maintain
  – Best to test concerns separately

• If the DOC is external, tests may be slow

Strategy 1: Try to avoid the DOC

• Write code that can be tested without a DOC
  – code that can be isolated from dependencies

• Some units are naturally independent
• Typically, utility library routines are like this

• E.g., a sorting routine
  – The only fixture needed is the input array

• Goal 1: create more independent units …
Refactoring to avoid the DOC: Extract Method

- Recall the bugaton demo refactoring:
  ```python
def bugaton():
    … get git log …
    … split log …
    … parse each message …
  
def bugaton():
      log = getGitLog()
      messages = splitLog(log)
      for m in messages:
        print parseMessage(m)

  
def getGitLog():
    … get git log …

def splitLog(log):
    … split log …

def parseMessage(m):
    … parse a message …
```

- Top-level functions should look more like pseudo-code
- Isolate complex functionality in easy to invoke functions
- Dependencies should be only on method parameters

Strategy 2: If can’t remove the DOC, use it

- First test Module1 as the SUT
- Then use it as a DOC to test Module2, etc.
- Disadvantages:
  - Tests get larger and larger, slower and slower
  - We want to have some unit tests for Module2 also

Strategy 3: If can’t remove the DOC, double it

- **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)
- Useful when DOC is too complex, or not yet available

Kinds of Test Doubles (Ch 23. UTP)

- The doubles can be hard-coded for one test, or configurable
Example: Hard-Coded Test Stub

• Recall the bugaton demo

```python
# Our SUT
def sutGitLog(docGit):
    log = docGit.cmd("log")
    … split log …

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

# Our test
def test_hard_coded_stub():
    # setup stub
    stub = GitStub()
    # exercise the SUT
    log = sutGitLog(stub)
    # verify
    assertEquals(log, …)

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

Example: Configurable Test Stub

```python
# Configurable stub
class GitConfigStub:
    self.reply = None
    # DOC interface methods
    def cmd(self, args):
        if self.exc:
            raise self.exc
        else:
            return self.reply
    # Configuration methods
    def setReply(self, val):
        self.reply = val
    def setException(self, val):
        self.exc = val

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

Example: Configurable Saboteur Test

```python
# Configurable stub
class GitConfigStub:
    self.reply = None
    self.exc = None
    # DOC interface methods
    def cmd(self, args):
        if self.exc:
            raise self.exc
        else:
            return self.reply
    # Configuration methods
    def setReply(self, val):
        self.reply = val
    def setException(self, val):
        self.exc = val

# Our test
def test_error():
    # create stub
    stub = GitConfigStub()
    # configure stub
    stub.setException(IOError("…"))
    # exercise the SUT
    log = sutGitLog(stub)
    # verify
    assertEquals(log, "error")
```

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
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)
      - See lecture materials on class page for comparison
    - pMock and Mox (for python)
      - etc.

Record-Replay Mocking Frameworks

- **Record-replay interface:**
  - Create the mock in record mode
  - Call the actual methods directly on the mock, who records them
  - Switch the mock to replay mode
  - Exercise the SUT (which indirectly calls the mock)
    - The mock checks that it sees the recorded calls

- **Examples:**
  - EasyMock (for Java), Mox (for Python)

Example: Test Mock Using Mox

```python
# Our test
def test_mock_1():
    # create mock in record mode
    mgit = mox.MockObject(Git)
    # record mode: call the methods we expect the SUT to call
    mgit.cmd("log --shortstat").AndReturn("...log value...")
    # Put all mocks in replay mode
    mox.ReplayAll()
    # Exercise test
    log = sutGitLog(mgit)
    # Verify test output
    assertEquals(log, "...log value...")
    # Verify indirect outputs, once SUT has finished the work
    mgit.VerifyAll()
```
Mock Errors

- **Unexpected method call**
  - E.g., If the SUT calls git with “log”
    Error: cmd("log") expected cmd("log --shortstat")

- **Missing method call**
  - If the SUT forgets to call git
    Error: missing expected cmd("log --shortstat")

Mock Errors

- **Unexpected method call**
  - E.g., If the SUT calls git with “log”
    Error: cmd("log") expected cmd("log --shortstat")

- **Missing method call**
  - If the SUT forgets to call git
    Error: missing expected cmd("log --shortstat")

# Our test
```python
def test_mock_2():
    # create mock in record mode
    mgit = mox.MockObject(Git)
    # record mode: call the methods we expect the SUT to call
    mgit.cmd("log --shortstat").AndReturn("...log value1...")
    # After first call, we expect 3 "log -p" in any order
    mgit.cmd("log -p rev1").InAnyOrder().AndReturn("...log value1...")
    mgit.cmd("log -p rev2").InAnyOrder().AndReturn("...log value2...")
    mgit.cmd("log -p rev3").InAnyOrder().AndReturn("...log value3...")
    # Put all mocks in replay mode
    mox.ReplayAll()
    # Exercise test
    log = sutGitLog(mgit)
    # Verify test output
    assertEquals(log, "...log value ...")
    # Verify indirect outputs
    mgit.VerifyAll()
```

# Our test
```python
def test_mock_3():
    # create mock in record mode
    mgit = mox.MockObject(Git)
    # record mock: call the methods we expect the SUT to call
    mgit.cmd(StrContains("--shortstat")).AndReturn("...log value1...")
    # After first call, we expect another call; don’t check args
    mgit.cmd(IgnoreArg()).AndReturn...
    # After second call, we expect some "log -p" in any order
    mgit.cmd(And(StrContains("-p"), StrContains("log"))).AndReturn...
    # Put all mocks in replay mode
    mox.ReplayAll()
    # Exercise test
    log = sutGitLog(mgit)
    # Verify test output
    assertEquals(log, "...log value ...")
    # Verify indirect outputs
    mgit.VerifyAll()
```

Mocks – In Any Order

- **Another family of mocking frameworks**
- **Test flow:**
  - Create mock
  - Set expectations on the mock
    - Expectations “read as specifications”
  - Exercise SUT
    - The mock checks that it sees the expected calls
- **Examples:**
  - jMock (for Java), Mock and pMock (for Python)
**Example: Test Mock Using pMock**

```python
# Our test
def test_mock_10():
    # create mock
    mgit = pMock.Mock()
    # setup the mock expectations
    mgit.expects(once()).cmd("log -shortstat")
        .will(return_value("...log value..."));
    # Exercise test
    log = sutGitLog(mgit)
    # Verify test output
    assertEquals(log, "...log value...")
    # Verify indirect outputs
    mgit.verify()
```

**Fluent Interface in jMock**

```java
// Define a state machine for the Mock
States gitApiStateMachineone = jMock.states("gapi").
    startsAs(INIT);
oneOf(mgit).cmd("log");
when(gitApiStateMachine.is(INIT));
    will(returnValue("..."));
then(gitApiStateMachine.is(AFTER_LOG));
atLeast(3).of(mgit).cmd(with(...));
when(gitApiStateMachine.is(AFTER_LOG));
    will(returnValue("..."));
then(gitApiStateMachine.is(INIT));
```

**Mocking for Statically-Typed Languages**

- Mocking frameworks use reflection to create mocks:
  - Run-time inspection of definition of classes
- If you mock an interface:
  - Framework will construct yet another implementation of the interface
- If you mock a class:
  - Can’t replace the class implementation
  - Framework will construct an adaptor (imposterizer)

**Fake Objects**

- A hand-written test double that implements (some of) the functionality of the DOC
- Useful when:
  - DOC is not yet available
  - DOC is across a major interface in the system
    - E.g., client/server
  - DOC is too heavy/slow/shared
    - E.g., database
    - E.g., web service
- Significant investment, but often worthwhile at the boundary between major components (and teams)
Example: Fake Objects

- Consider a system that monitors air temperature over time and triggers alerts

```
# A fake object to double TimeProvider
class FakeTimeProvider:
    # TimeProvider interface
    def getTime(self):
        # Return the current fake time
        return self.time
    def registerOnTick(onTick):
        ...as in real TimeProvider...
    # Configuration interface
    def startFake(self, start, end):
        self.time = start
        while self.time < end:
            self.time += 1
        ...call onTick handlers...
```

Example: Testing with Actual DOC

- DOC might not be available
  - E.g., not every dev. has the temperature sensors
- Tests will be slow
  - Have to wait for actual time to pass (e.g., for time outs)
- Can’t test exceptional scenarios
  - Extreme temperature readings
  - At midnight on Daylight-Savings-Time change
- Tests not reproducible
  - Hard to reproduce external conditions
  - Impossible to reproduce actual time exactly

```
# A fake object to double TempProvider
class FakeTempProvider:
    # TempProvider interface
    def getTemp(self):
        # Return the current fake temp
        return self.temp
    # Configuration interface
    def fakeStep(self, timeProvider, start, rate, end):
        timeProvider.registerOnTick(self.onTick)
        self.temp = start
        self.rate = rate
        self.end = end
    def onTick(self):
        if self.temp < self.end:
            self.temp += self.rate
```

Example: A FakeTimeProvider

- A fake object has two interfaces:
  1. One that is similar to the DOCs
  2. One that is for configuration/testing
- Every loop iteration simulates a second
- Tests are now synchronous and fast
- Can simulate many seconds in 1 second of actual test

Example: A FakeTempProvider

- A fake object has two interfaces:
  1. One that is similar to the DOCs
  2. One that is for configuration/testing
- Every loop iteration simulates a second
- Tests are now synchronous and fast
- Can simulate many seconds in 1 second of actual test
Doubles for Databases

- A database is a slow DOC:
  - 1 simple unit test in Django (about 20 SQL queries) takes 8000 msec with mysql
- The usual abstraction for a DB is SQL queries
- Replace the DB with sqlite3, for tests
  - Almost same SQL (framework can bridge the gap)
  - Same test is 150 msec with SQLite3
- If you can use a higher-level hash map abstraction
  - Same test is now 10 msec
  - Works only in very simple scenarios (e.g., warmup)

Databases and Testing

- A database is a heavy and delicate DOC:
  - Should not use the production database!
  - Typically in a separate process, over TCP, using disk, enforcing concurrency control, transactions
  - Relatively slow for unit tests
- Frameworks provide test support for databases
  - Use special test databases (on same server)
  - Clear the database before each test
  - Can use other DB engines (e.g., sqlite3 for tests)
- Learn the testing support for your framework