

Software Testing (3)

September 27, 2022

Byung-Gon Chun

(Slide credits: George Candea, EPFL and Armando Fox, UCB)

Test Double

Test Double for Unit Tests

- Prerequisite: a project constructed in a well-disciplined manner, i.e. using Dependency Injection
- Various names for stubs, mocks, fakes, dummies, and other things that people use to stub out parts of a system for testing
=> Test double

Test Double

- **Fake** objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an InMemoryTestDatabase is a good example).
- **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
- **Spies** are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.
- **Mocks** are pre-programmed with expectations which form a specification of the calls they are expected to receive. They can throw an exception if they receive a call they don't expect and are checked during verification to ensure they got all the calls they were expecting.

The Need for Test Double (interchangeably Mocking)

- Removing external dependencies from a unit test in order to create a controlled environment around it
 - We mock all other classes that interact with the class that we want to test
- Why?
 - Increased speed
 - Avoiding undesired side effects during testing

The Need for Mocking

- Common targets for mocking
 - Database connections
 - Web services
 - Classes that are slow
 - Classes with side effects
 - Classes with non-deterministic behavior
 - ...

The Need for Mocking

- Think of a class that communicates with an external payment provider
 - No need to actually connect to the payment provider each time the unit test runs.
 - It would be dangerous to test code that charges credit cards using a live payment environment.
 - It would also make the unit test non-deterministic, e.g. if the payment provider is down for some reason.

Mocking

- Fake classes that replace these external dependencies
- Instructed before the test starts to behave as you expect

Stubs

- A stub is a fake class that comes with *preprogrammed return values*
- It's injected into the class under test to give you absolute control over what's being tested as input.
- A typical stub is a database connection that allows you to mimic any scenario without having a real database.

Mocks

- A mock is a fake class that *can be examined after the test is finished* for its interactions with the class under test
- For example, you can ask it whether a method was called or how many times it was called
- Typical mocks are classes with side effects that need to be examined, e.g. a class that sends emails or sends data to another external service.

Mockito

- A popular mocking framework for JUnit tests
- Uses the term “mocks” for everything
- Stubbing directive:
 `when(something).thenReturn(somethingElse)`
 `when(something).thenAnswer(new Answer() { ... })`
 // Generic interface to be used for configuring mock's answer.
- Simple mocking directive:
 `verify(something).methodName(...)`

Mocking for Unit Tests

- Stubbing method calls

// you can mock concrete classes, not only interfaces

```
LinkedList mockedList = mock(LinkedList.class); // a fake class object
```

// stubbing appears before the actual execution

```
when(mockedList.get(0)).thenReturn("first");
```

// the following actual execution prints "first"

```
System.out.println(mockedList.get(0));
```

// the following prints "null" because get(999) was not stubbed

```
System.out.println(mockedList.get(999));
```

Mocking for Unit Tests

```
import static org.mockito.Mockito.*;

// mock creation
List mockedList = mock(List.class); // mock an interface

// using the mock object
mockedList.add("one");
mockedList.clear();

// selective, explicit, highly readable verification
verify(mockedList).add("one");
verify(mockedList).clear();
```

How Do We Test CustomerReader?

```
public class Customer {  
    private long id;  
    private String firstName;  
    private String lastName;  
    //...getters and setters  
    // redacted for brevity...  
}
```

Naïve approach

pre-fill a real database with customers and run this test against it. This is problematic for a lot of reasons. It creates a **hard dependency on a running database**, and also requires an extra step to create the test data.

```
public class CustomerReader {  
    private EntityManager entityManager;  
  
    public String findFullName(Long customerID) {  
        // ... code here ...  
        Customer customer = entityManager.find(Customer.class, customerID);  
        return customer.getFirstName() + " " + customer.getLastName();  
    }  
}
```

How Do We Test CustomerReader?

```
public class Customer {  
    private long id;  
    private String firstName;  
    private String lastName;  
    //...getters and setters  
    // redacted for brevity...  
}
```

The best solution for a true unit test is to **completely remove the database dependency**.

We will **stub** the database connection instead, and "fool" our class to think that it is talking to a real EntityManager, while in reality the EntityManager is a Mockito stub

```
public class CustomerReader {  
    private EntityManager entityManager;  
  
    public String findFullName(Long customerID) {  
        // ... code here ...  
        Customer customer = entityManager.find(Customer.class, customerID);  
        return customer.getFirstName() + " " + customer.getLastName();  
    }  
}
```

Stubbing to Test CustomerReader

```
public class CustomerReaderTest {  
  
    @Test  
    public void happyPathScenario() {  
        Customer sampleCustomer = new Customer();  
        sampleCustomer.setFirstName("Susan");  
        sampleCustomer.setLastName("Ivanova");  
  
        EntityManager entityManager = mock(EntityManager.class);  
        when(entityManager.find(Customer.class, 1L)).thenReturn(sampleCustomer);  
  
        // dependency injection  
        CustomerReader customerReader = new CustomerReader();  
        customerReader.setEntityManager(entityManager);  
  
        String fullName = customerReader.findFullName(1L);  
        assertEquals("Susan Ivanova", fullName);  
    }  
}
```


Multiple Test Methods

```
public class CustomerReaderTest {  
    //Class to be tested  
    private CustomerReader customerReader;  
    //Dependencies  
    private EntityManager entityManager;  
  
    @Before  
    public void setup(){  
        customerReader = new CustomerReader();  
  
        entityManager = mock(EntityManager.class);  
        customerReader.setEntityManager(entityManager);  
        // dependency injected  
    }  
    ...  
}
```

Multiple Test Methods

```
public class CustomerReaderTest {  
    ...  
  
    @Test  
    public void happyPathScenario(){  
        Customer sampleCustomer = new Customer();  
        sampleCustomer.setFirstName("Susan");  
        sampleCustomer.setLastName("Ivanova");  
  
        when(entityManager.find(Customer.class, 1L)).thenReturn(sampleCustomer);  
  
        String fullName = customerReader.findFullName(1L);  
        assertEquals("Susan Ivanova", fullName);  
    }  
    ...  
}
```

Multiple Test Methods

```
public class CustomerReaderTest {  
    @Test  
    public void customerNotPresentInDb() {  
        when(entityManager.find(Customer.class, 1L)).thenReturn(null);  
  
        String fullName = customerReader.findFullName(1L);  
        assertEquals("", fullName);  
    }  
}
```

Mock Example

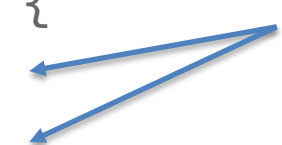
As soon as you try to write a unit test for this class, you will notice that **nothing can really be asserted**.

The method that we want to test - `notifyIfLate` - is a void method that cannot return anything. So how do we test it?

```
public class LateInvoiceNotifier {  
  
    private final EmailSender emailSender;  
    private final InvoiceStorage invoiceStorage;  
  
    public LateInvoiceNotifier(final EmailSender emailSender,  
                               final InvoiceStorage invoiceStorage) {  
        this.emailSender = emailSender;  
        this.invoiceStorage = invoiceStorage;  
    }  
  
    public void notifyIfLate(Customer customer) {  
        if(invoiceStorage.hasOutstandingInvoice(customer)) {  
            emailSender.sendEmail(customer);  
        }  
    }  
}
```

Mock Example

```
public class LateInvoiceNotifier {  
  
    private final EmailSender emailSender;  
    private final InvoiceStorage invoiceStorage;  
  
    public LateInvoiceNotifier(final EmailSender emailSender,  
                               final InvoiceStorage invoiceStorage) {  
        this.emailSender = emailSender;  
        this.invoiceStorage = invoiceStorage;  
    }  
  
    public void notifyIfLate(Customer customer) {  
        if(invoiceStorage.hasOutstandingInvoice(customer)) {  
            emailSender.sendEmail(customer);  
        }  
    }  
}
```



two external dependencies

In this case, we need to focus on the side effects of the code.
The side effect here is sending an email.

This email is sent only if an outstanding invoice is present.

Mocking Example

```
public class LateInvoiceNotifierTest {
```

```
    //Class to be tested
```

```
    private LateInvoiceNotifier lateInvoiceNotifier;
```

```
    //Dependencies (will be mocked)
```

```
    private EmailSender emailSender;
```

```
    private InvoiceStorage invoiceStorage;
```

```
    //Test data
```

```
    private Customer sampleCustomer;
```

```
@Before
```

```
public void setup(){
```

```
    invoiceStorage = mock(InvoiceStorage.class);
```

```
    emailSender = mock(EmailSender.class);
```

```
    lateInvoiceNotifier = new LateInvoiceNotifier(emailSender, invoiceStorage);
```

```
    sampleCustomer = new Customer();
```

```
    sampleCustomer.setFirstName("Susan");
```

```
    sampleCustomer.setLastName("Ivanova");
```

```
}
```

```
...
```

Mocking Example

(contain no assertion, use verify)

```
public class LateInvoiceNotifierTest {
    @Test
    public void lateInvoice() {
        when(invoiceStorage.hasOutstandingInvoice(sampleCustomer)).thenReturn(true);

        lateInvoiceNotifier.notifyIfLate(sampleCustomer);

        verify(emailSender).sendEmail(sampleCustomer);
    }

    @Test
    public void noLateInvoicePresent() {
        when(invoiceStorage.hasOutstandingInvoice(sampleCustomer)).thenReturn(false);

        lateInvoiceNotifier.notifyIfLate(sampleCustomer);

        verify(emailSender, times(0)).sendEmail(sampleCustomer);
    }
}
```

Python Mock Library: unittest.mock

- *unittest.mock* provides a class called *Mock* which you will use to imitate *real objects* in your codebase
- *patch()* replaces the real objects in your code with Mock instances. You can use *patch()* as either a decorator or a context manager, giving you control over the scope in which the object will be mocked.

Python Mock Example

```
class Calculator:
    def sum(self, a, b):
        return a + b

from unittest import TestCase
from main import Calculator

class TestCalculator(TestCase):
    def setUp(self):
        self.calc = Calculator()

    def test_sum(self):
        answer = self.calc.sum(2, 4)
        self.assertEqual(answer, 6)
```

```
python -m unittest
.
```

Ran 1 test in 0.003s

OK

Python Mock Example

```
import time

class Calculator:
    def sum(self, a, b):
        time.sleep(1000) # long running process
        return a + b

from unittest import TestCase
from unittest.mock import patch

class TestCalculator(TestCase):
    @patch('main.Calculator.sum', return_value=9)
    def test_sum(self, sum):
        self.assertEqual(sum(2,3), 9)
```

Python Mock Example

```
from unittest import TestCase
from unittest.mock import patch
import time
```

```
def sum(a, b):
    time.sleep(1000)
    return a + b
```

```
def mock_sum(a, b):
    # mock sum function without the long running time.sleep
    return a + b
```

```
class TestCalculator(TestCase):
    @patch('main.Calculator.sum', side_effect=mock_sum)
    def test_sum(self, sum):
        self.assertEqual(sum(2,3), 5)
        self.assertEqual(sum(7,3), 10)
```

Run a custom sum function



Python Mock Example

```
import requests
```

```
class Blog:
```

```
    def __init__(self, name):  
        self.name = name
```

```
    def posts(self):
```

```
        response = requests.get("https://jsonplaceholder.typicode.com/posts")  
        return response.json()
```

```
    def __repr__(self):
```

```
        return '<Blog: {}>'.format(self.name)
```

Unpredictable API call



Python Mock Example

```
from unittest import TestCase
from unittest.mock import patch, Mock
```

```
class TestBlog(TestCase):
    @patch('main.Blog')
    def test_blog_posts(self, MockBlog):
        blog = MockBlog()
        blog.posts.return_value = [
            {
                'userId': 1,
                'id': 1,
                'title': 'Test Title',
                'body': 'Far out in the uncharted backwaters of the
unfashionable end of the western spiral arm of the Galaxy lies a small
unregarded yellow sun.'
            }
        ]
```

```
        response = blog.posts()
        self.assertIsNotNone(response)
        self.assertIsInstance(response[0], dict)
```

Calling `blog.posts()` on our mock blog object returns our predefined JSON

Python Mock Example

```
>>> from unittest.mock import Mock
```

```
>>> # Create a mock object ...
```

```
json = Mock()
```

```
>>> json.loads('{"key": "value"}')
```

```
<Mock name='mock.loads()' id='4550144184'>
```

```
>>> # You know that you called loads() so you can
```

```
>>> # make assertions to test that expectation ...
```

```
json.loads.assert_called()
```

Another Python Testing Framework Option

- Pytest
 - fixtures
- monkeypatch: helper methods for safely patching and mocking functionality in pytest tests

Specification-style Testing

Spec-style Testing

- BDD: (high-level) story-based testing (*will be covered in the requirements lecture)
- Spec-style testing: use functional specs
- describe()/beforeEach()/it() convention originated with the Ruby testing library Rspec
- Alternative to basic unit testing tools like xUnit
- E.g.
 - Jest: Javascript unit testing (multiple layers on top of Jasmine)
 - Jasmine: behavior driven development (BDD) testing framework for JavaScript

Frontend Testing Tools We Use

- Jest
 - JavaScript testing tool
 - suites, specs, expectations, matchers
 - mocks and spies
- Enzyme
 - JavaScript testing library for React
 - render, access, and manipulate DOM elements

Jest

- describe, it or test
- expect, matcher (toEqual, ...)
- beforeEach, afterEach, beforeAll, afterAll
- jest.fn, jest.spyOn

Jest/Jasmine

- Test suites: a collection of tests

```
describe('Tweet utilities module', function () {  
  // Suite implementation goes here...  
});
```

- Spec: individual test

```
it('returns an array of tweet ids', function () {  
  // Spec implementation goes here...  
});  
test('returns an array of tweet ids', function () {  
  // Spec implementation goes here...  
});
```

Jest/Jasmine

- Expectation and Matcher

`expect(actualListOfTweetIds).toEqual(expectedListOfTweetIds);`

- Matchers
- `.toEqual(value)`
 - `.toBe(value)`
 - `.toHaveBeenCalled()`
 - `.toHaveBeenCalledTimes(number)`
 - `.toHaveBeenCalledWith(arg1, arg2, ...)`
 - `.toHaveBeenLastCalledWith(arg1, arg2, ...)`
 - `.toHaveBeenNthCalledWith(nthCall, arg1, arg2,)`
 - `.toHaveReturned()`
 - `.toHaveReturnedTimes(number)`
 - `.toHaveReturnedWith(value)`
 - ...

Jest/Jasmine

- Setups

- Repeating setup

```
beforeEach(() => {  
    initializeCityDatabase();  
});
```

```
afterEach(() => {  
    clearCityDatabase();  
});
```

- One-time setup

```
beforeAll(() => {  
    initializeCityDatabase();  
});
```

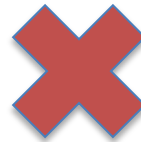
```
afterAll(() => {  
    clearCityDatabase();  
});
```

Testing Asynchronous Code

- When you have code that runs asynchronously, Jest needs to know when the code it is testing has completed, before it can move on to another test.
- Callback

fetchData(callback) function fetches some data and calls callback(data) when it is complete. You want to test that this returned data is the string 'peanut butter'.

```
test('the data is peanut butter', () => {  
  function callback(data) {  
    expect(data).toBe('peanut butter');  
  }  
  fetchData(callback);  
});
```



Testing Asynchronous Code

- **Callback**

fetchData(callback) function fetches some data and calls callback(data) when it is complete. You want to test that this returned data is the string 'peanut butter'.

Jest will wait until the done is called before finishing the test.

- **Promise**

fetchData returns a promise that is supposed to resolve to the string 'peanut butter'

Jest will wait for that promise to resolve.

```
test('the data is peanut butter', done => {  
  function callback(data) {  
    try {  
      expect(data).toBe('peanut butter');  
      done();  
    } catch (error) {  
      done(error);  
    }  
  }  
  fetchData(callback);  
});
```

```
test('the data is peanut butter', () => {  
  return fetchData().then(data => {  
    expect(data).toBe('peanut butter');  
  });  
});
```


Mock

- `jest.fn()`

```
const myMock = jest.fn();  
console.log(myMock());  
// > undefined
```

```
myMock  
  .mockReturnValueOnce(10)  
  .mockReturnValueOnce('x')  
  .mockReturnValue(true);
```

```
console.log(myMock(), myMock(), myMock(), myMock());  
// > 10, 'x', true, true
```

```
const stubTodoList = [{  
  id: 0,  
  title: 'title 1',  
  content: 'content 1'  
}, ];
```

```
// Replace axios.get with mock  
axios.get = jest.fn(url => {  
  return new Promise((resolve, reject) =>  
  {  
    const result = {  
      status: 200, data: stubTodoList  
    };  
    resolve(result);  
  })  
});
```

Enzyme: Testing for React

- shallow: shallow rendering
- mount: full rendering
- shallow vs. mount
- wrapper.find(<selector>): find every node in the render tree that matches the provided selector
- simulate: simulate events
- ...

Enzyme Example

```
// App.js
const App = () => (
  <div foo="bar">
    <div>Hello world</div>
  </div>
);
export default App;
```

```
// App.js
import Bar from './bar';
const Foo = () => <div>Foo!</div>;

const App = () => (
  <div>
    <Foo />
    <Bar />
  </div>
);
```

```
// App.test.js
import { shallow } from 'enzyme';
const wrapper = shallow(<App />);
console.log(wrapper.debug());
```

```
<div foo="bar">
  <div>Hello world</div>
</div>
```

```
<div>
  <Foo />
  <Bar />
</div>
```

Enzyme Example

```
import React from 'react';
import { expect } from 'chai';
import { shallow } from 'enzyme';
import sinon from 'sinon';
import MyComponent from './MyComponent';
import Foo from './Foo';
```

...

```
describe('<MyComponent />', () => {
  it('renders three <Foo /> components', () => {
    const wrapper = shallow(<MyComponent />);
    expect(wrapper.find(Foo)).toHaveLength(3);
  });
});
```

...

```
it('simulates click events', () => {
  const onClick = sinon.spy();
  const wrapper = shallow(<Foo onClick={onClick} />);
  wrapper.find('button').simulate('click');
  expect(onClick).toHaveBeenCalled();
});
```

sinon: standalone test spies, stubs
and mocks for JavaScript.

chai is an *assertion library like Jest*