Test Scaffolding

Giuseppe Scanniello

Simone Romano

Michelangelo Esposito

Test Scaffolding

Any code used to facilitate testing activities

- It is not part of the “product” as seen by the end user

It includes **test drivers** and **test doubles**

Purpose of test scaffolding

Increase controllability and observability

- **Controllability:** ability to affect the system behavior (in particular, replicate that behavior)
  - How easy it is to provide a system with the needed inputs, in terms of values, operations, and behaviors
- **Observability:** ability to observe the system behavior
  - How easy it is to observe the behavior of a system in terms of its outputs, effects on the environment, and other hardware and software components

Dependencies make controllability and observability difficult

How to affect the behavior of the SUT?

The behavior of the **SUT---System Under Test---**can be affected directly, i.e., via its "front door" (e.g., public API), or indirectly, i.e., via its "back door"

Direct inputs

Values a test inject into the SUT via its front door

Indirect inputs

When the behavior of the SUT is affected by the values returned by another component whose services it uses, we call these values indirect inputs of the SUT

How to observe the behavior of the SUT?

The behavior of the SUT can be observed thanks to its direct or indirect outputs

Direct outputs

- Responses a test receives from the SUT via its front door
  - Return values of method calls, updated arguments passed by reference, exceptions raised, …

Indirect outputs

- When the behavior of the SUT includes actions that cannot be observed through the public API of the SUT but which are seen/experienced by other components, we call these actions indirect outputs of the SUT
  - Method calls to another component, messages sent on a message channel, records inserted into a database/file, …

Testability

The degree to which a system facilitates the establishment of test criteria, and the performance of tests, to determine whether these criteria have been met

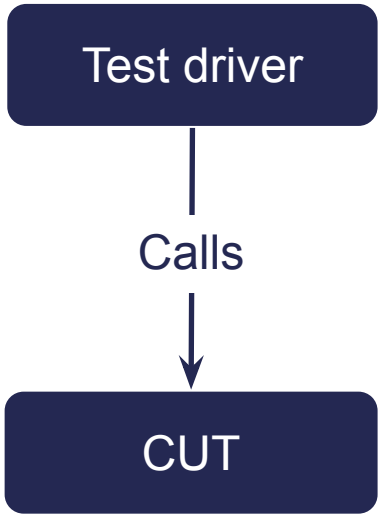
- How hard it is to find faults in a system

Low controllability and observability implies low testability

Test driver

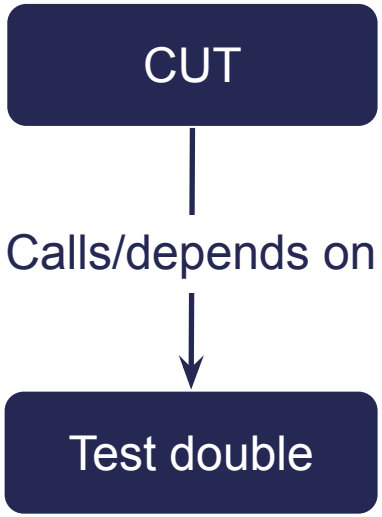
A component that replaces the calling component of a **CUT---Component Under Test**

A test method implements a driver since it replaces the actual calling component



Test double

A component that replaces a component (aka **depended-on component---DOC**) on which the CUT depends



When should I use test doubles?

DOCs **not available** yet at testing time

- E.g., I have to test a CUT A that depends on a component B, however B has not been implemented yet; in order to test A, I replace B with a test double (e.g., a stub)

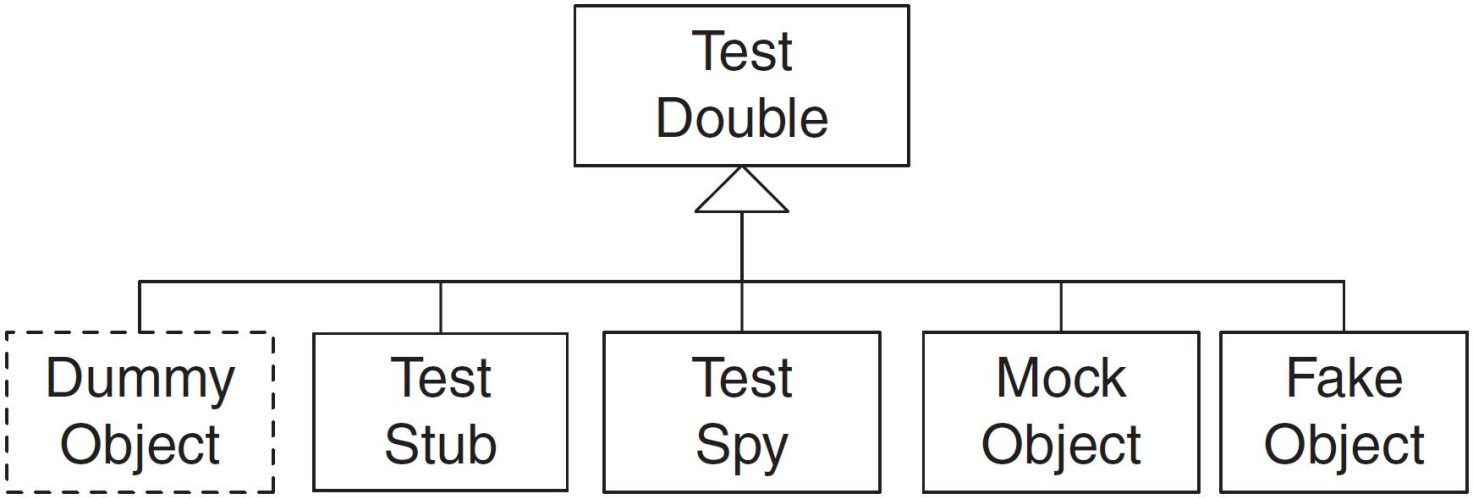
DOCs **too expensive** in terms of time or resources so it is preferable to avoid their use during unit testing

Make test execution deterministic so removing side effect due to DOCs

...

Taxonomy of test doubles

It has been proposed by Gerard Meszaros [2007]



Dummy object

A replacement for an object that, although required, is never used by the CUT

- E.g., a dummy object can be used to fill the parameter list of a method where a parameter is never used

It does nothing, i.e., its methods have either no implementation or just throw an exception if called

Motivating example for a dummy object

A lot of code needed just to set up the fixture

```
def test_invoice_add_line_item(self):
    QUANTITY = 1
    product = Product(get_unique_number_as_string(), get_unique_number())
    state = State('West Dakota', 'WD')
    city = City('Centreville', state)
    address = Address('123 Blake St.', city, '12345')
    customer = Customer(get_unique_number_as_string(), get_unique_number(),
        address)
    invoice = Invoice(customer)

    # Exercise
    invoice.add_item_quantity(product, QUANTITY)

    # Verify
    line_items = invoice.get_line_items()
    self.assertEqual(1, len(line_items), 'number of items')
    actual = line_items[0]
    exp_item = LineItem(invoice, product, QUANTITY)
    self.assertLineItemsEqual(actual, exp_item)
```

A `Customer` object is required but never used

Solution with a dummy object

```
def test_invoice_add_line_item(self):
    QUANTITY = 1
    product = Product(get_unique_number_as_string(), get_unique_number())
    invoice = Invoice(DummyCustomer())
    exp_item = LineItem(invoice, product, QUANTITY)

    # Exercise
    invoice.add_item_quantity(product, QUANTITY)

    # Verify
    line_items = invoice.get_line_items()
    self.assertEqual(1, len(line_items), 'number of items')
    actual = line_items[0]

    self.assertLineItemsEqual(actual, exp_item)
```

Simpler fixture

Dummy object as parameter

Dummy object implementation

```
class DummyCustomer(ICustomer):
    def __init__(self):
        pass

    def get_zone(self):
        raise RuntimeError('This should never be called!')
```

Test stub

A replacement for a DOC that injects the desired **indirect inputs** into the CUT

- It enables the control of the indirect inputs

A test stub used to inject **valid** indirect inputs is a **responder**

- A responder test stub is used in *happy* path testing

A test stub used to inject **invalid** indirect inputs is a **saboteur**

- It returns unexpected values\objects or raises exceptions

A test stub that replaces a DOC not yet available is called **temporary**

- Once the actual DOC is available it replaces the temporary test stub

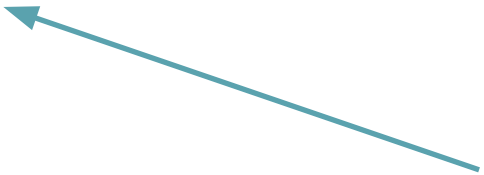
Test stub

Two ways of implementing a test stub:

- **Hard-coded test stub**
  - It returns hard-coded values
  - It is purpose-built for a single test or a very small number of tests
- **Configurable test stub**
  - It allows avoiding building a different hard-coded test stub for each test
  - Need of configuring its behavior in the fixture setup phase

Motivating example for a responder

```
def test_display_current_time_at_midnight(self):
    sut = TimeDisplay()
    # Exercise
    result = sut.get_current_time_ass_html_fragment()
    # Verify
    expected_time = '<span class="tinyBoldText">Midnight</span>'
    self.assertEqual(expected_time, result)
```



It verifies an HTML string containing the current time at midnight---unfortunately, this test rarely passes because it depends on the real system clock

Solution with an hard-coded responder

```
def test_display_current_time_at_midnight(self):
    test_stub = MidnightTimeProvider()
    sut = TimeDisplay()
    sut.set_time_provider(test_stub)
    # Exercise
    result = sut.get_current_time_as_html_fragment()
    # Verify
    expected_time = '<span class="tinyBoldText">Midnight</span>'
    self.assertEqual(expected_time, result, 'Midnight')
```

Test stub configuration

Test stub installation

Test stub implementation

```
class MidnightTimeProvider(TimeProvider):
    def get_time(self):
        my_time = new Calendar()
        my_time.set(Calendar.HOUR_OF_DAY, 0)
        my_time.set(Calendar.MINUTE, 0)
        return my_time
```

Solution with a configurable responder

```
def test_display_current_time_at_midnight(self):
    tp_stub = TimeProviderTestStub()
    tp_stub.set_hours(0)
    tp_stub.set_minutes(0)
    sut = TimeDisplay()
    sut.set_time_provider(test_stub)
    # Exercise
    result = sut.get_current_time_as_html_fragment()
    # Verify
    expected_time = '<span class="tinyBoldText">Midnight</span>'
    self.assertEqual(expected_time, result, 'Midnight')
```

Test stub configuration

Test stub installation

Test stub implementation

```
class TimeProviderTestStub(TimeProvider):
    def __init__(self):
        self.calendar = Calendar()

    def set_hours(self, hours):
        self.calendar.set(Calendar.HOUR_OF_DAY, 0)

    def set_minutes(self, minutes):
        self.calendar.set(Calendar.MINUTE, 0)

    def get_Time():
        return self.calendar
```

```
        return myTime;
    }
}
```

Test spy

It can be seen as a more sophisticated test stub with some recording capabilities

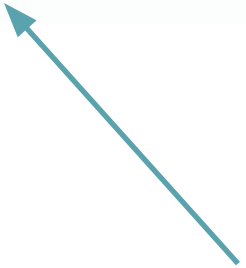
- When testing a CUT, it **also** enables the observation of the indirect outputs

It replaces a DOC and records the calls to its methods when it is exercised by a test, then that test compares the actual values recorded by the test spy with the expected values

It can be implemented in either an **hard-coded** or **configurable** way

Motivating example for a test spy

```
def test_remove_flight(self):
    expected_flight_dto = create_registered_flight()
    facade = FlightManagementFacade()
    # Exercise
    facade.remove_flight(expected_flight_dto.get_flight_number())
    # Verify
    self.assertFalse('flight should not exist after being removed',
        facade.flight_exists(expected_flight_dto.get_flight_number()))
```



It verifies the removal of a flight but does not verify the indirect outputs---i.e., the fact that a DOC is expected to log each time a flight is removed along with the date and username of the requester

Solution with a test spy

```python
def test_remove_flight(self):
    expected_flight_dto = create_registered_flight()
    facade = FlightManagementFacade()
    log_spy = AuditLogSpy()
    facade.set_audit_log(log_spy)
    # Exercise
    facade.remove_flight(expected_flight_dto.get_flight_number())
    # Verify
    self.assertFalse('flight should not exist after being removed',
        facade.flight_exists(expected_flight_dto.get_flight_number()))
    self.assertEqual(1, log_spy.get_number_of_calls(), 'number of calls')
    self.assertEqual(Helper.REMOVE_FLIGHT_ACTION_CODE,
        log_spy.get_action_code(), 'action code')
    self.assertEqual(helper.get_todays_date(), log_spy.get_date(), 'date')
    self.assertEqual(helper.TEST_USER_NAME, log_spy.get_user(), 'user')
    self.assertEqual(expected_flight_dto.get_flight_number(),
        log_spy.get_detail(), 'detail')
```

Test spy installation

Verify direct output

Verify indirect outputs

Test spy implementation

```python
class AuditLogSpy:
    date
    user, action_code
    detail
    number_of_calls

    def log_message(self, date,
     user, action_code, detail):
        self.date = date
        self.user = user
        self.action_code = action_code
        self.detail = detail
        self.number_of_calls += 1

    def get_number_of_calls(self):
        return self.number_of_calls

    def get_date(self):
        return self.date

    def get_user(self):
        return self.user

    def get_action_code(self):
        return self.action_code

    def get_detail(self):
        return self.detail
```

Fake object

A replacement for a DOC that implements the same functionality of that DOC but in a much simpler way

Typical usage: when the use of the actual DOC would make the tests slow (e.g., database or web service)

It can be also used when a DOC is not available yet

Fake objects vs test stubs:

Stubs do not actually implement DOCs' functionality---they just returns hard-coded or configured values

Fake objects do implement DOCs' functionality in a lighter-weight way

Motivating example for a fake object

```
def test_read_write(self):
    facade = FlightManagementFacade()
    yyc = facade.create_airport('YYC', 'Calgary', 'Calgary')
    lax = facade.create_airport('LAX', 'LAX Intl', 'LA')
    facade.create_flight(yyc, lax)
    # Exercise
    flights = facade.get_flights_by_origin_airport(yyc)
    # Verify
    self.assertEqual(1, len(flights), '# of flights')
    flight = flights[0]
    self.assertEqual('yyc', flight.get_origin().get_code(), 'origin')
```

It calls `create_airport` (twice), which calls the data access layer so causing the test to slow down

```
def create_airport(self, airport_code,
    name, nearby_city):
    transaction_manager.begin_transaction()
    airport = data_access.create_airport(airport_code, name, nearby_city)
    log_message('Wrong anction code', airport_code.get_action_code)
    transaction_manager.commit_transaction()
    return airport_get_id()
```

Solution with a fake object

```
def test_read_write(self):
    facade = FlightManagementFacade()
    facade.set_dao(InMemoryDatabase())
    yyc = facade.create_airport('YYC', 'Calgary', 'Calgary')
    lax = facade.create_airport('LAX', 'LAX Intl', 'LA')
    facade.create_flight(yyc, lax)
    # Exercise
    flights = facade.get_flights_by_origin_airport(yyc)
    # Verify
    self.assertEqual(1, len(flights), '# of flights')
    flight = flights[0]
    self.assertEqual('yyc', flight.get_origin().get_code(), 'origin')
```

Fake object installation

```
class InMemoryDatabase(FlightDao):
    def create_airport(self, airport_code,
        name, nearby_city):
        assert_parameters_are_valid(airport_code, name, nearby_city)
        assert_airport_doesnt_exist(airport_code)
        result = Airport(airport_code, name, nearby_city)
        self.airports.append(result)
        return result
```

Fake object implementation---it is an in-memory database that is faster than the actual DOC

Mock object

A replacement for a DOC that provides a **dummy implementation** for that DOC and enables both **control** of the indirect inputs and **observation** of the indirect outputs

- It can be seen as a more sophisticated test spy with **further** capabilities

Dis/similarities with test stub and spy:

- Like a test stub, a mock object enables the control of the indirect inputs
- Like a test spy, a mock object enables the observation of the indirect outputs
- Unlike a test spy, when comparing actual calls received with the previously defined expectations, it performs assertions and fails the test on behalf of the test method

Mock object

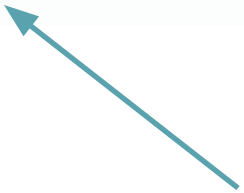
Two variants of mock objects:

- **Strict** mock object---it fails the test if *correct* calls are received in a different order than was specified
- **Lenient** mock object---it tolerates out-of-order calls (some lenient mock objects tolerate or even ignore unexpected calls or missed calls)

It can be implemented in either an **hard-coded** or **configurable** way

Motivating example for a mock object

```
def test_remove_flight(self):
    expected_flight_dto = create_registered_flight()
    facade = FlightManagementFacade()
    # Exercise
    facade.remove_flight(expected_flight_dto.get_flight_number())
    # Verify
    self.assertFalse('flight should not exist after being removed',
        facade.flight_exists(expected_flight_dto.get_flight_number()))
```



You should remember this example, it is the same as the test spy

Solution with a mock object

```
def test_remove_flight(self):
    expected_flight_dto = create_registered_flight()
    mock_log = ConfigurableMockAuditLog()
    mock_log.set_expected_log_message(helper.get_todays_date(), Helper.TEST_USER_NAME,
        Helper.REMOVE_FLIGHT_ACTION_CODE, expected_flight_dto.get_flight_number())
    mock_log.set_expected_number_calls(1)
    facade = FlightManagementFacade()
    facade.set_audit_log(mock_log)
    # Exercise
    facade.remove_flight(expected_flight_dto.get_flight_number())
    # Verify
    self.assertFalse('flight should not exist after being removed',
        facade.flight_exists(expected_flight_dto.get_flight_number()))
    mock_log.verify()


class ConfigurableMockAuditLog(AuditLog):
    expected_date
    expected_user, expected_action_code
    expected_detail
    expected_number_calls = 1
    actual_number_calls = 0

    def log_message(self, actual_date, actual_user, actual_action_code,
        actual_detail, actual):
        self.actual_number_calls += 1
        Assert.assertEqual(self.expected_date, actual_date, 'date')
        Assert.assertEqual(self.expected_user, actual_user, 'user')
        Assert.assertEqual(self.expected_action_code, actual_action_code, 'action_code')
        Assert.assertEqual(self.expected_detail, actual_detail, 'detail')

    def verify(self):
        Assert.assertEqual(self.expected_number_calls, self.actual_number_calls,
            'number of calls')
```

Mock object installation

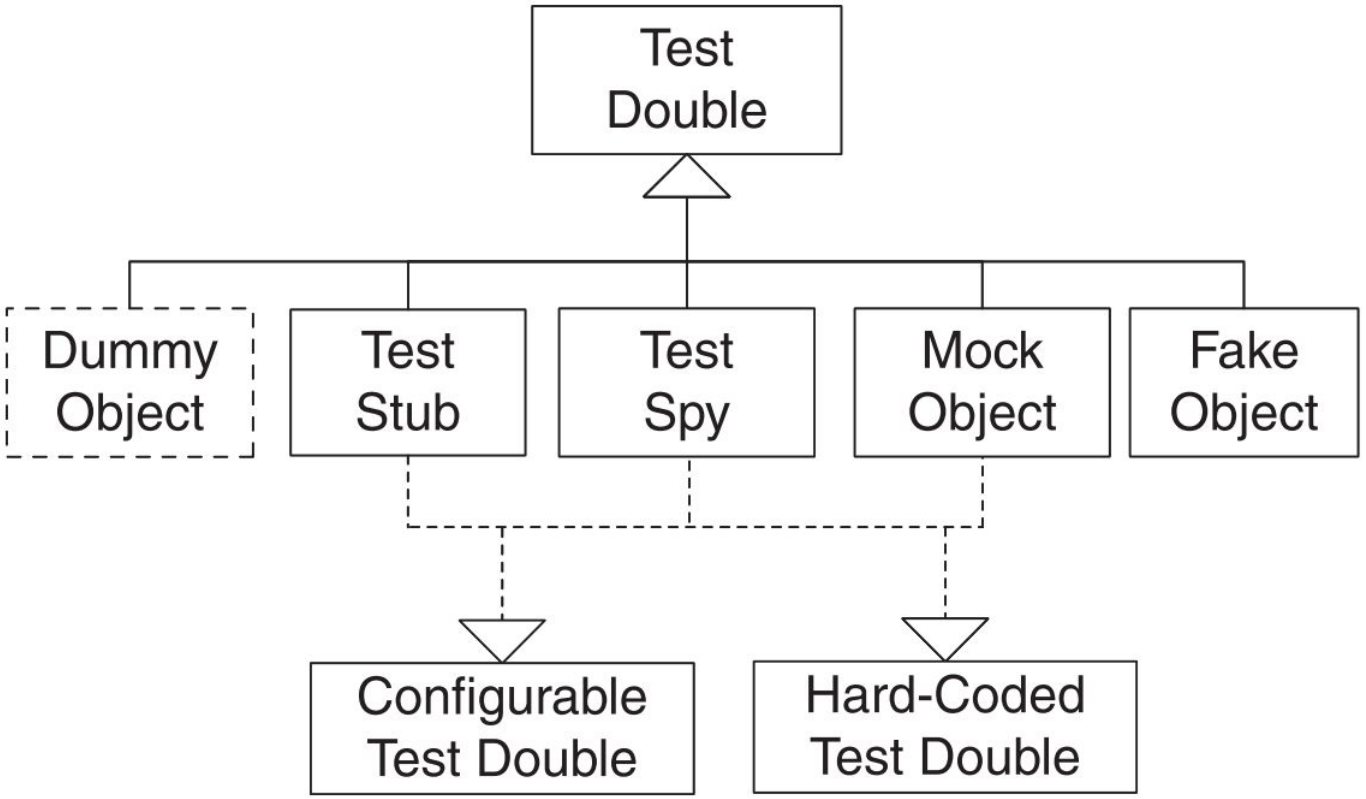
Verify direct output

Mock object implementation

It verifies indirect outputs when it is called

It verifies the call to `logMessage()` was actually made

Summary of test doubles



Test code refactoring

The previous motivating examples can be seen as **test anti-patterns**

- If a test method looks like one the motivating example, then you should start thinking to refactor that method using the corresponding suggested solutions (e.g., fake object, test spy, etc.)

Implementing test doubles

So far we have seen how to manually implement the various types of test double.

Some frameworks (i.e. unittest) provide some tools to facilitate this process.

Mocking with unittest

```python
import unittest
from unittest.mock import patch
import TemperatureSensor
from Thermostat import Thermostat

class ThermostatTest(unittest.TestCase):

    @patch('TemperatureSensor.read_temperature')
    def test_low_temperature_activation_1(self, mock_temperature):
        mock_temperature.return_value = 15
        temperature = TemperatureSensor.read_temperature()

        # Thermostat requires a certain temperature range to activate
        thermostat = Thermostat()
        result = thermostat.activate(temperature)
        self.assertTrue(result)

    @patch.object(TemperatureSensor, 'read_temperature')
    def test_low_temperature_activation_2(self, mock_temperature):
        mock_temperature.return_value = 15
        temperature = TemperatureSensor.read_temperature()

        # Thermostat requires a certain temperature range to activate
        thermostat = Thermostat()
        result = thermostat.activate(temperature)
        self.assertTrue(result)
```

Functions have not yet been implemented

```python
def read_temperature():
    raise NotImplementedError


def read_humidity():
    raise NotImplementedError
```

Path of the function to mock

Mock object parameter

Setting the return value for the mocked function

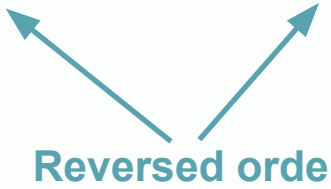
Alternative syntax

Mocking with unittest

IMPORTANT: when using multiple decorators on your test method, the mapping on the parameters **works backwards**.

```
@patch.object(TemperatureSensor, 'read_temperature')
@patch.object(TemperatureSensor, 'read_humidity')
def test_low_temperature_humidity_activation(self, mock_humidity, mock_temperature):
    mock_temperature.return_value = 10
    mock_humidity.return_value = 5
    temperature = TemperatureSensor.read_temperature()
    humidity = TemperatureSensor.read_humidity()

    thermostat = Thermostat()
    result = thermostat.activate(temperature, humidity)
    self.assertTrue(result)
```



Reference book