

Agile Software Development

Produced
by

Eamonn de Leastar (edelestar@wit.ie)

Department of Computing, Maths & Physics
Waterford Institute of Technology

<http://www.wit.ie>

<http://elearning.wit.ie>



Waterford Institute of Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

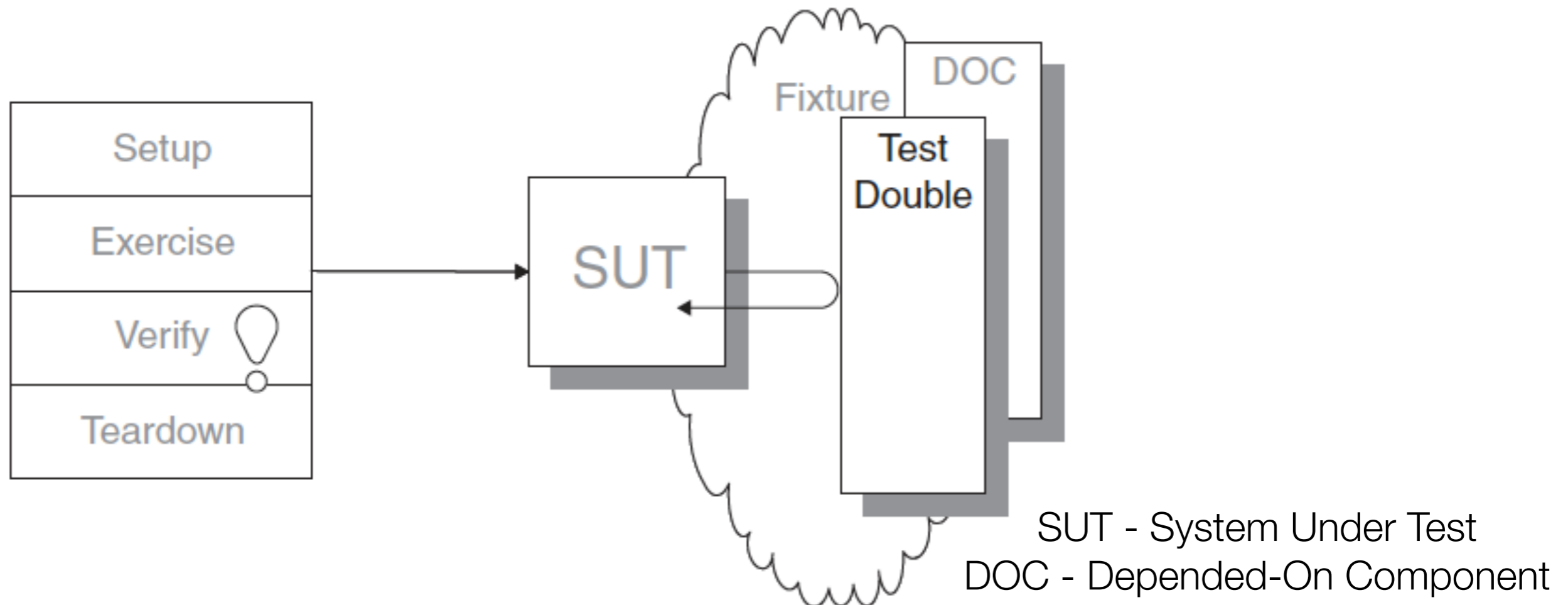


Test Double

How can we verify logic independently when code it depends on is unusable?

How can we avoid Slow Tests?

We replace a component on which the SUT depends with a “test-specific equivalent.”



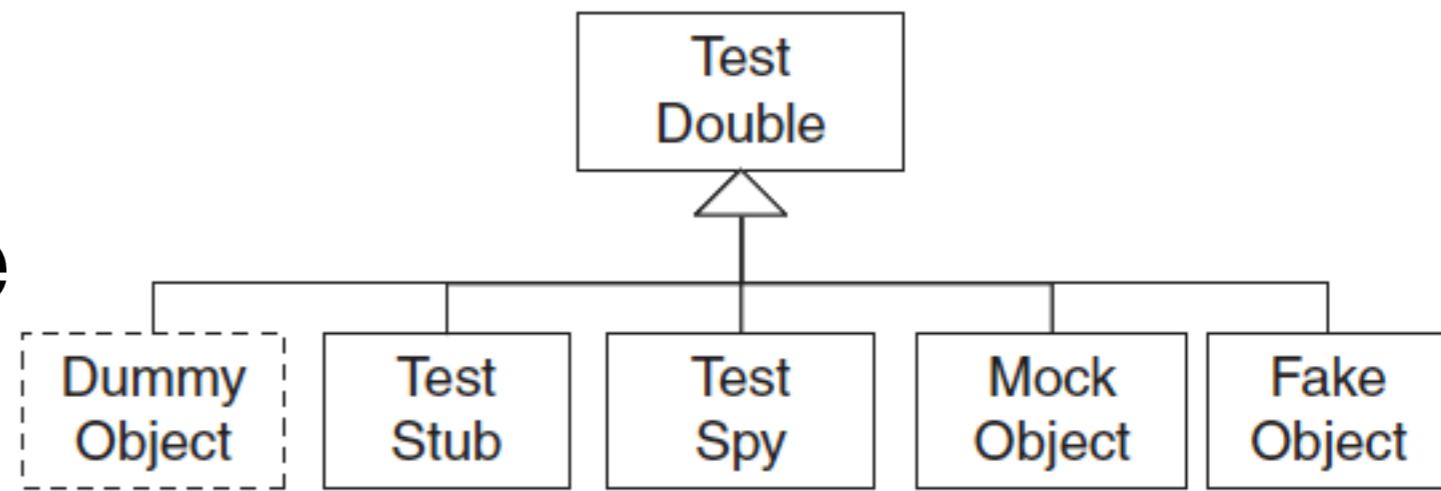
What is a Test Double?

- Hard to test the SUT because it depends on other components that cannot be used in the test environment
- Eg - those components aren't available, will not return the results needed for the test, or executing them would have undesirable side effects
- When writing a test in which we cannot use the real Depended-on Component (DOC), we can replace it with a Test Double.
- The Test Double doesn't have to behave exactly like DOC, it merely has to provide the same API so that the SUT thinks it is the real one
- Called after "Stunt Double" in movie making - the stunt person takes the place of the real actor

When to use it

- If we have an untested requirement because neither the SUT nor its DOCs provide an observation point for the SUT's indirect output that we need to verify
- If we have untested code and a DOC does not provide the control point to allow us to exercise the SUT with the necessary indirect inputs
- If we have slow tests and we want to be able to run our tests more quickly and hence more often

Variations of Test Double

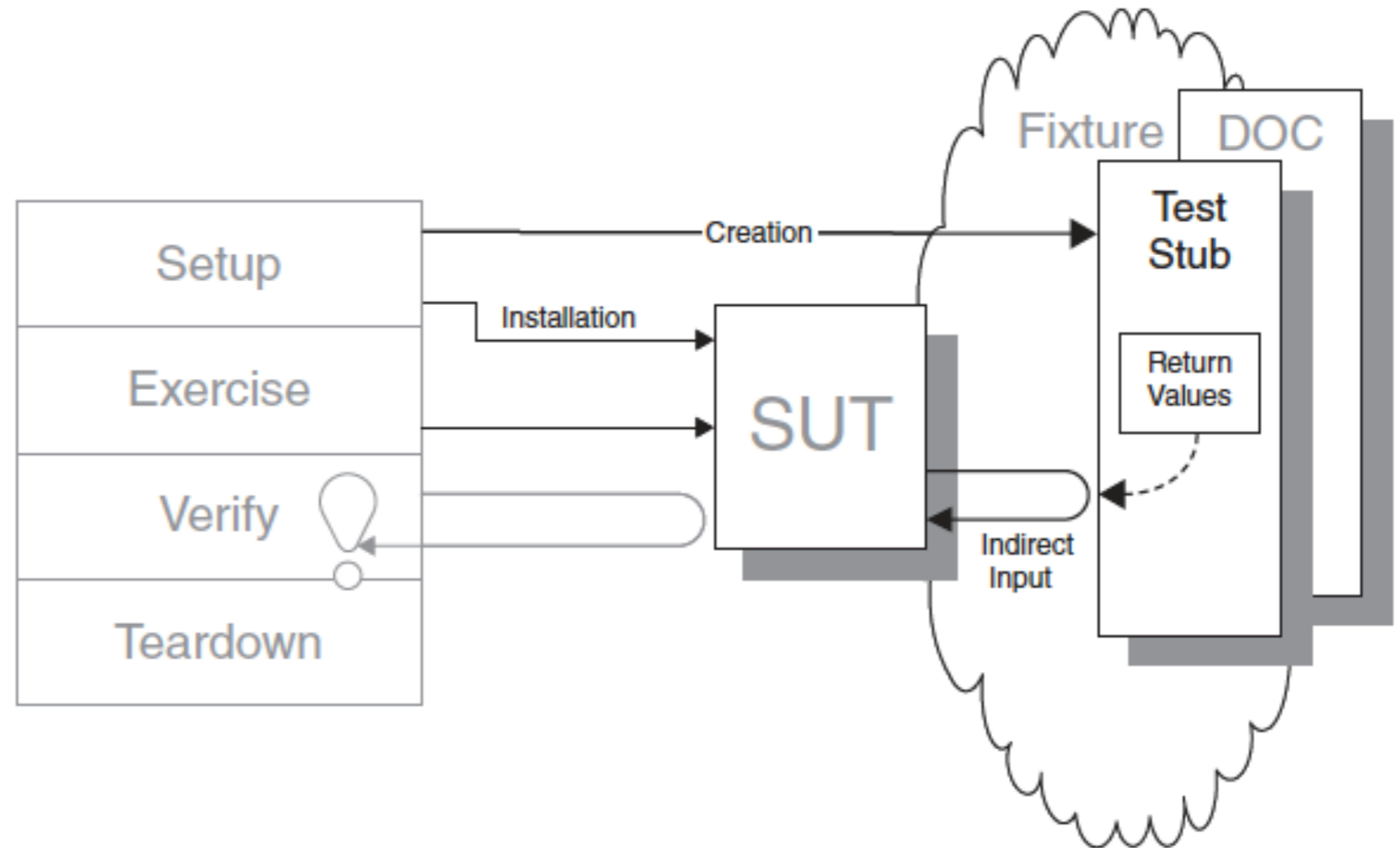


- *Dummy Object*: Some method signatures of the SUT may require objects as parameters. If neither the test nor the SUT cares about these objects, pass in a null object reference or an instance of the Object class.
- *Test Stub*: Replace a real component on which the SUT depends so that the test has a control point for the indirect inputs of the SUT. No behaviour simulated or values returned.
- *Test Spy*: a more capable version of a Test Stub - provide values to the SUT in response to method calls.
- *Mock Object*: an observation point to verify the indirect outputs of the SUT as it is exercised. Typically, it includes the functionality of a Test Stub in that it must return values to the SUT, but the emphasis is on the verification of the indirect outputs.
- *Fake Object*: replace the functionality of a real DOC in a test for reasons other than verification of indirect inputs and outputs of the SUT. Perhaps real object too slow, has undesirable side effects etc...

Test Stub

How can we verify logic independently when it depends on indirect inputs from other software components?

We replace a real object with a test-specific object that feeds the desired indirect inputs into the system under test.



Test Stub Motivation

```
public void testDisplayCurrentTime_AtMidnight()
{
    // fixture setup
    TimeDisplay sut = new TimeDisplay();
    // exercise SUT
    String result = sut.getCurrentTimeAsHtmlFragment();
    // verify direct output
    String expectedTimeString = "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals( expectedTimeString, result);
}
```

- Verifies the basic functionality of a component that formats an HTML string containing the current time.
- Depends on the real system clock so it rarely ever passes!

Test Stub Example

```
public void testDisplayCurrentTime_AtMidnight()
{
    // Fixture setup
    // Test Double configuration
    TimeProvider tpStub = new TimeProviderTestStub();
    tpStub.setHours(0);
    tpStub.setMinutes(0);
    // Instantiate SUT
    TimeDisplay sut = new TimeDisplay();
    // Test Double installation
    sut.setTimeProvider(tpStub);
    // Exercise SUT
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Verify outcome
    String expectedTimeString = "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals("Midnight", expectedTimeString, result);
}
```

- Note that TimeDisplay (SUT) depends on TimeProvider (DOC).
- The DOC is replaced with a stub - TimeProviderTestStub which is hand coded to return 00:00 time.

Test Stub Using JMock Library

```
public void testDisplayCurrentTime_AtMidnight_JM()
{
    // Fixture setup
    TimeDisplay sut = new TimeDisplay();
    // Test Double configuration
    Mock tpStub = mock(TimeProvider.class);
    Calendar midnight = makeTime(0,0);
    tpStub.stubs().method("getTime").withNoArguments().will(returnValue(midnight))
    // Test Double installation
    sut.setTimeProvider((TimeProvider) tpStub);
    // Exercise SUT
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Verify outcome
    String expectedTimeString = "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals("Midnight", expectedTimeString, result);
}
```

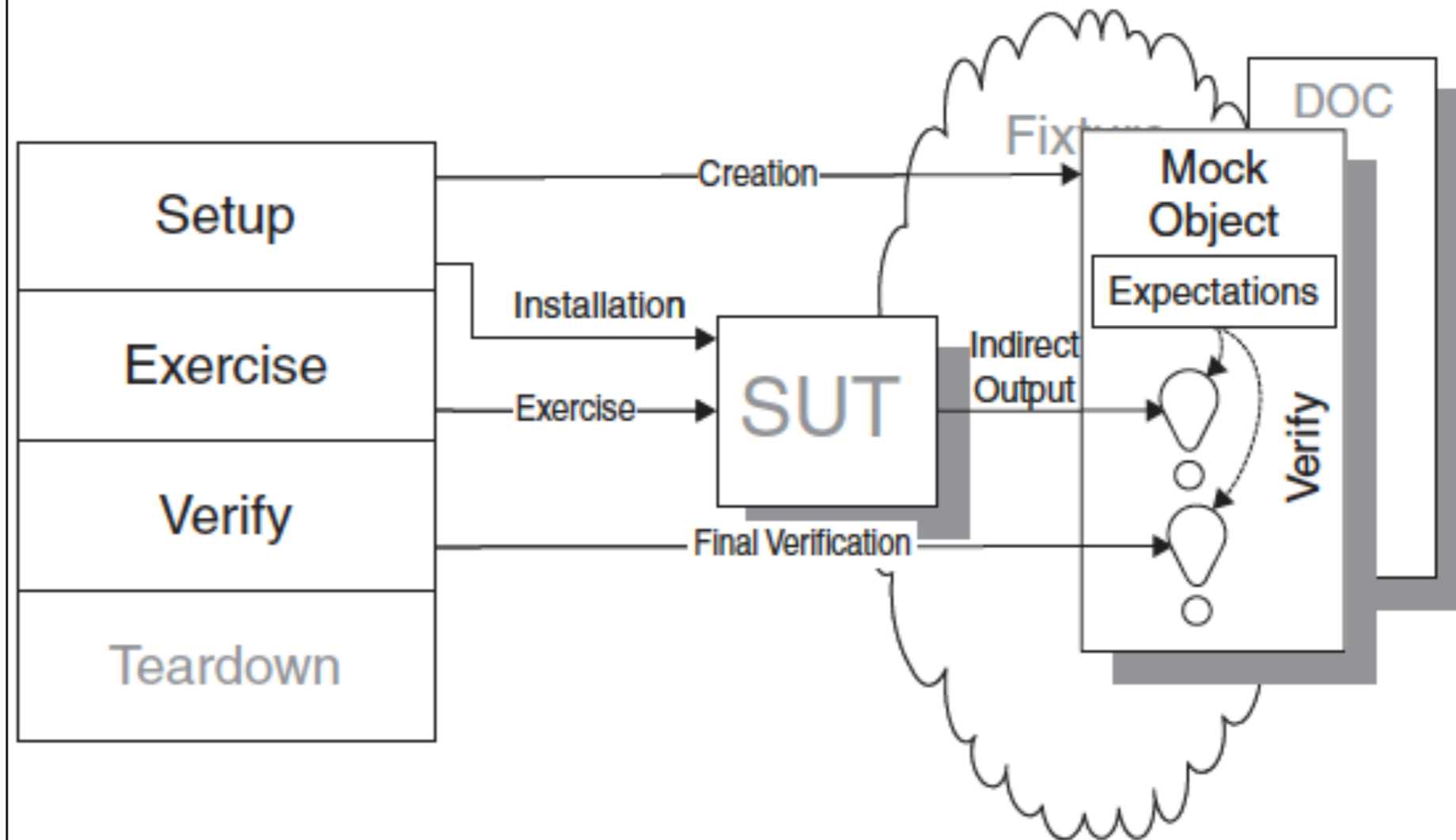
- There is no Test Stub implementation to examine for this test because the JMock framework implements the Test Stub using reflection

Mock Object

How do we implement Behavior Verification for indirect outputs of the SUT?

How can we verify logic independently when it depends on indirect inputs from other software components?

We replace an object on which the SUT depends on with a test specific object that verifies it is being used correctly by the SUT.



How it Works

- Define a Mock Object that implements the same interface as an object on which the SUT depends.
- During the test, configure the Mock Object with the values with which it should respond to the SUT and the method calls (complete with expected arguments) it should expect from the SUT.
- Before exercising the SUT, install the Mock Object so that the SUT uses it instead of the real implementation.
- When called during SUT execution, the Mock Object compares the actual arguments received with the expected arguments using equality assertions and fails the test if they don't match.

Implementation

- Tests written using Mock Objects look different from more traditional tests because all the expected behavior must be specified before the SUT is exercised.
- This makes the tests harder to write and to understand.
- The standard Four-Phase Test is altered somewhat when we use Mock Objects.
- In particular, the fixture setup phase of the test is broken down into three specific activities and the result verification phase more or less disappears, except for the possible presence of a call to the “final verification” method at the end of the test.

Test Structure

- Fixture setup:
 - Test *constructs* Mock Object.
 - Test *configures* Mock Object.
 - Test *installs* Mock Object into SUT.
 - Test sets *expectations* on mock object. i.e. what behavior it expects to be triggered by SUT
- Exercise SUT:
 - SUT calls Mock Object; Mock Object does assertions.
- Result verification:
 - Test calls “final verification” method.
- Fixture teardown:
 - No impact.

Example -Motivation

(from JMock Documentation)

- A Publisher sends messages to zero or one Subscriber.
- We want to test the Publisher, which involves testing its interactions with its Subscribers.
- We will test that a Publisher sends a message to a single registered Subscriber.
- To test interactions between the Publisher and the Subscriber we will use a mock Subscriber object

```
public interface Subscriber
{
    void receive(String message);
}
```

```
public class Publisher
{
    private Subscriber subscriber;

    public void add(Subscriber subscriber)
    {
        this.subscriber = subscriber;
    }

    public void publish(String message)
    {
        if (subscriber != null)
            subscriber.receive(message);
    }
}
```

Configure Test Case

- First we must import the jMock classes, define our test fixture class and create a "Mockery" that represents the context in which the Publisher exists.
- The context mocks out the objects that the Publisher collaborates with (in this case a Subscriber) and checks that they are used correctly during the test.

```
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.integration.junit4.JMock;
import org.jmock.integration.junit4.JUnit4Mockery;
import org.junit.Test;
import org.junit.runner.RunWith;

import sut.Publisher;
import sut.Subscriber;

@RunWith(JMock.class)
public class PublisherTest
{
    Mockery context = new JUnit4Mockery();
    ...
}
```

Fixture Setup (1)

- Write the method that will perform our test - first set up the context in which our test will execute:
 - *Construct*: create a Publisher to test.
 - *Configure*: create a mock Subscriber that should receive the message.
 - *Install*: register the Subscriber with the Publisher.

```
@Test
public void oneSubscriberReceivesAMessage()
{
    Publisher publisher = new Publisher();
    final Subscriber subscriber = context.mock(Subscriber.class);
    publisher.add(subscriber);
    ....
}
```


Fixture Setup (2)

- Define *expectations* on the mock Subscriber that specify the methods that we expect to be called upon it during the test run.
- We expect the receive method to be called once with a single argument, the message that will be sent.

```
@Test
public void oneSubscriberReceivesAMessage()
{
    ...

    context.checking(new Expectations()
    {{
        oneOf (subscriber).receive(message);
    }});

    ...
}
```

Exercise SUT

- We then execute the code that we want to test.

```
@Test
public void oneSubscriberReceivesAMessage()
{
    ...
    publisher.publish(message);
    ...
}
```

Result Verification

- After the code under test has finished our test must verify that the mock Subscriber was called as expected.
- If the expected calls were not made, the test will fail. The `MockObjectTestCase` does this automatically.
- You don't have to explicitly verify the mock objects in your tests.

```
@Test
public void oneSubscriberReceivesAMessage()
{
    Publisher publisher = new Publisher();
    final Subscriber subscriber
        = context.mock(Subscriber.class);
    publisher.add(subscriber);

    final String message = "message";
    context.checking(new Expectations()
    {{
        oneOf (subscriber).receive(message);
    }});

    publisher.publish(message);
}
```

The jMock Cookbook

How to...

1. [Get Started](#)
2. [Define Expectations](#)
3. [Return Values from Mocked Methods](#)
4. [Throw Exceptions from Mocked Methods](#)
5. [Match Parameter Values](#)
6. [Precisely Specify Expected Parameter Values](#)
7. [Expect Methods More \(or Less\) than Once](#)
8. [Expect a Sequence of Invocations](#)
9. [Expect an Invocation Between Two Other Invocations](#)
10. [Ignore Irrelevant Mock Objects](#)
11. [Override Expectations Defined in the Test Set-Up](#)
12. [Match Objects and Methods](#)
13. [Write New Matchers](#)
14. [Write New Actions](#)
15. [Easily Define Actions with Scripts](#)
16. [Test Multithreaded Code with Mock Objects](#)
17. [Mock Generic Types](#)
18. [Mock Abstract and Concrete Classes](#)
19. [Use jMock with Languages Other Than Java](#)
20. [Upgrade from jMock 1 to jMock 2](#)
21. [Use jMock in Maven Builds](#)
22. [Understand method dispatch in jMock 2](#)
23. [Mock Classes in Eclipse Plug-in Tests](#)
24. [Mock asynchronous GWT services](#)

Using jMock from Maven Builds

- The jMock 2 jars are accessible via Maven 2 by declaring the following dependencies in your POM.
- All the required dependencies on jMock will be included automatically.

```
<dependency>  
  <groupId>org.jmock</groupId>  
  <artifactId>jmock-junit4</artifactId>  
  <version>2.5.1</version>  
</dependency>
```

Returning Values

- Return values from mocked methods by using the `returnValue` action within the "will" clause of an expectation.

```
oneOf (calculator).add(2, 2); will(returnValue(5));
```

- The `returnIterator` action returns an iterator over a collection.

```
final List<Employee> employees = new ArrayList<Employee>();  
employees.add(alice);  
employees.add(bob);  
  
context.checking(new Expectations()  
{  
    {  
        oneOf (department).employees(); will(returnIterator(employees));  
    }  
});
```

- A convenient overload of the `returnIterator` method lets you specify the elements inline:

```
context.checking(new Expectations()  
{  
    {  
        oneOf (department).employees(); will(returnIterator(alice, bob));  
    }  
});
```

Exceptions

- Use the `throwException` action to throw an exception from a mocked method.

```
allowing (bank).withdraw(with(any(Money.class)));  
    will(throwException(new WithdrawalLimitReachedException()));
```

Expecting Methods More (or Less) than Once

- **oneOf** The invocation is expected once and once only.
- **exactly(n).of** The invocation is expected exactly n times. Note: one is a convenient shorthand for exactly(1).
- **atLeast(n).of** The invocation is expected at least n times.
- **atMost(n).of** The invocation is expected at most n times.
- **between(min, max).of** The invocation is expected at least min times and at most max times.
- **allowing** The invocation is allowed any number of times but does not have to happen.
- **ignoring** The same as allowing. Allowing or ignoring should be chosen to make the test code clearly express intent.
- **never** The invocation is not expected at all. This is used to make tests more explicit and so easier to understand.

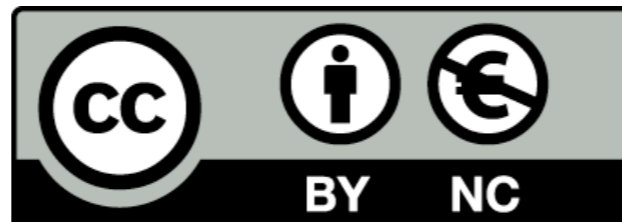
Mocking Classes with jMock and the ClassImposteriser

- Because it uses Java's standard reflection capability, the default configuration of the jMock framework can only mock interfaces, not classes.
- However, the ClassImposteriser extension class uses the [CGLIB 2.11](#) and [Objenesis2](#) libraries to create mock objects of classes as well as interfaces.
- This is useful when working with legacy code to tease apart dependencies between tightly coupled classes.

```
import org.jmock.Mockery;
import org.jmock.Expectations;
import org.jmock.integration.junit4.JUnit4Mockery;
import org.jmock.lib.legacy.ClassImposteriser;

@RunWith(JMock.class)
public class ConcreteClassTest
{
    private Mockery context = new JUnit4Mockery()
    {
        {
            setImposteriser(ClassImposteriser.INSTANCE);
        }
    };

    @Test
    void someTest()
    {
        Graphics g = context.mock(java.awt.Graphics.class);
        // expectations and tests
    }
}
```



Except where otherwise noted, this content is licensed under a Creative Commons Attribution-NonCommercial 3.0 License.

For more information, please see <http://creativecommons.org/licenses/by-nc/3.0/>

