Agile Software Development

Produced by

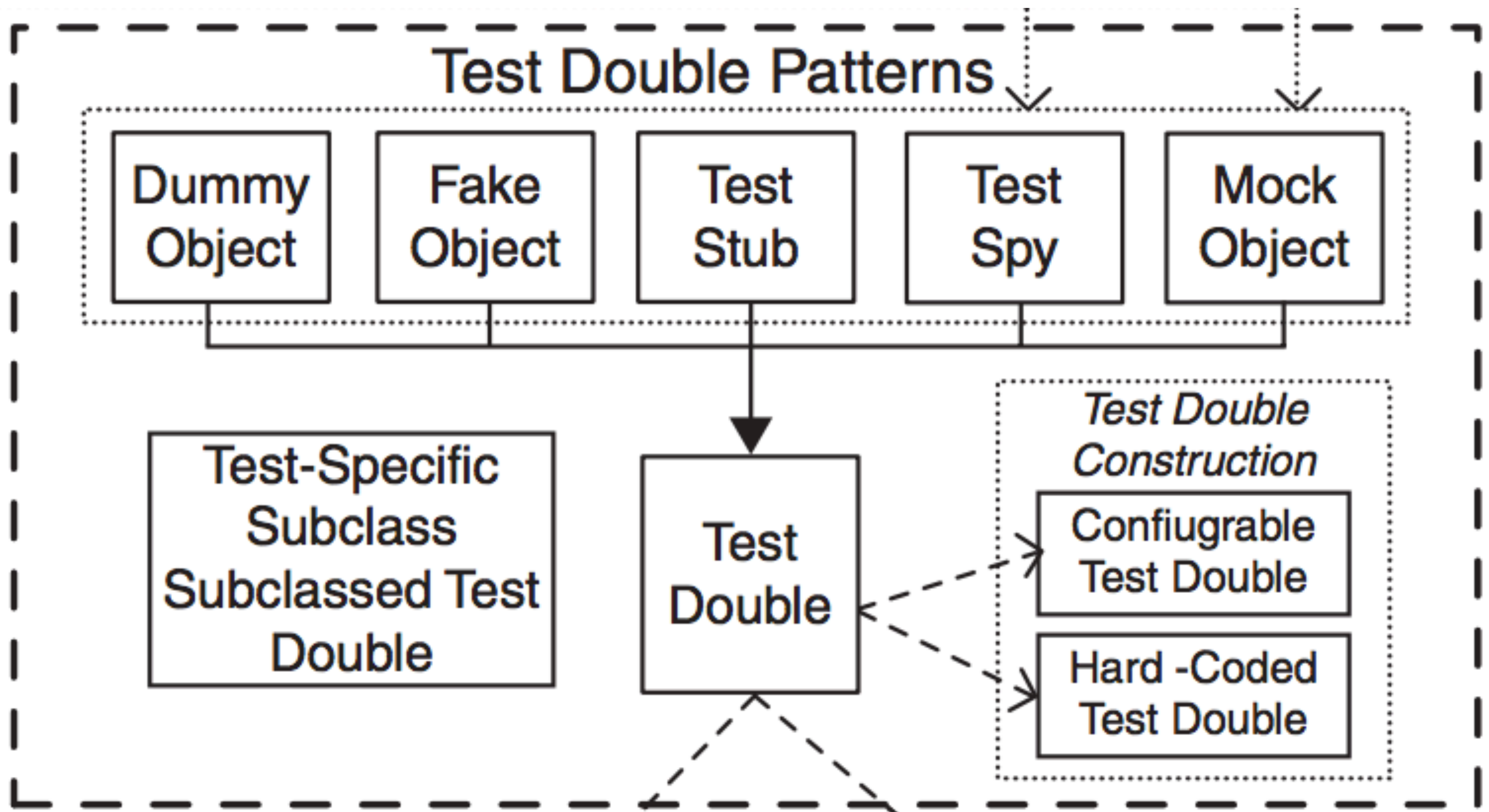
Eamonn de Leastar (edeleastar@wit.ie)

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

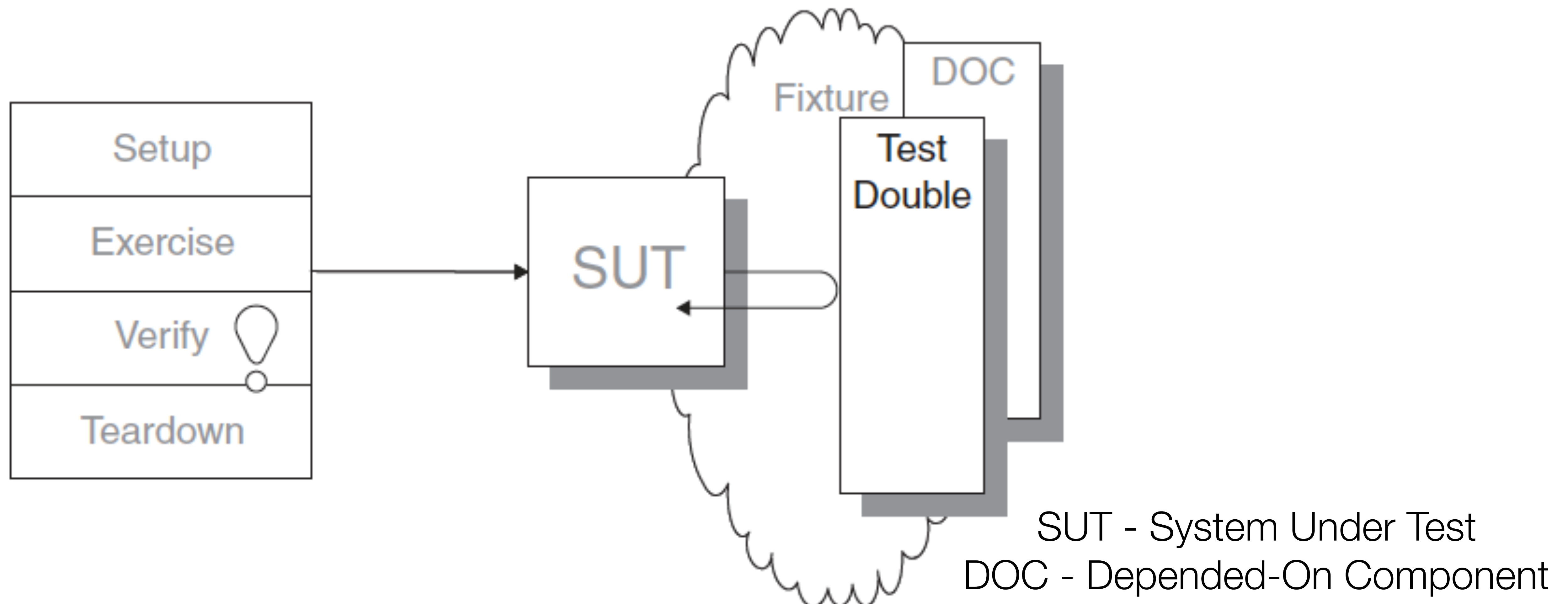

Test Double Patterns
Dummy Object
Fake Object
Test Stub
Test Spy
Mock Object
Test-Specific Subclass Subclassed Test Double
Test Double
Test Double Construction
Confiugrable Test Double
Hard -Coded Test Double

Test Double

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

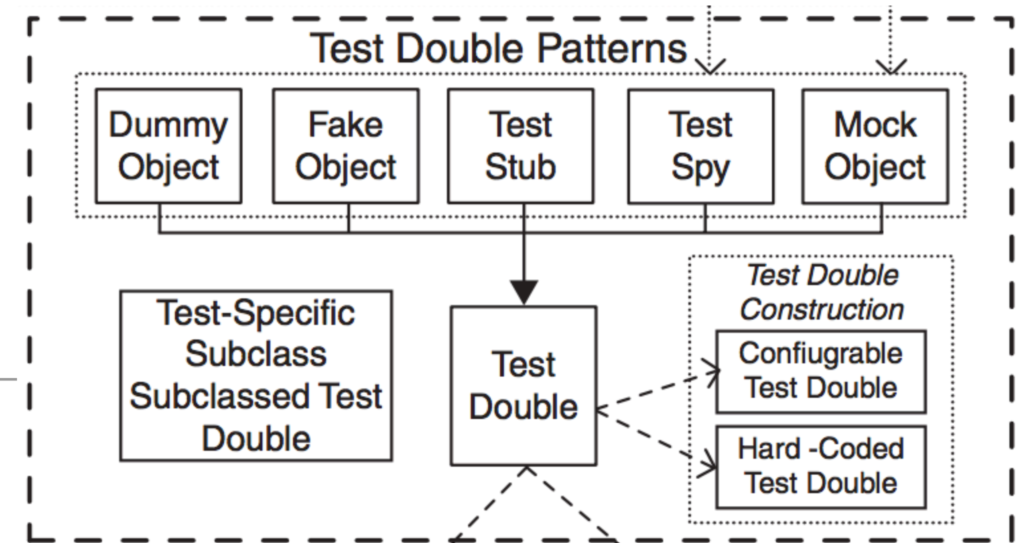
How can we avoid Slow Tests?

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



SUT - System Under Test
DOC - Depended-On Component

What is a Test Double?

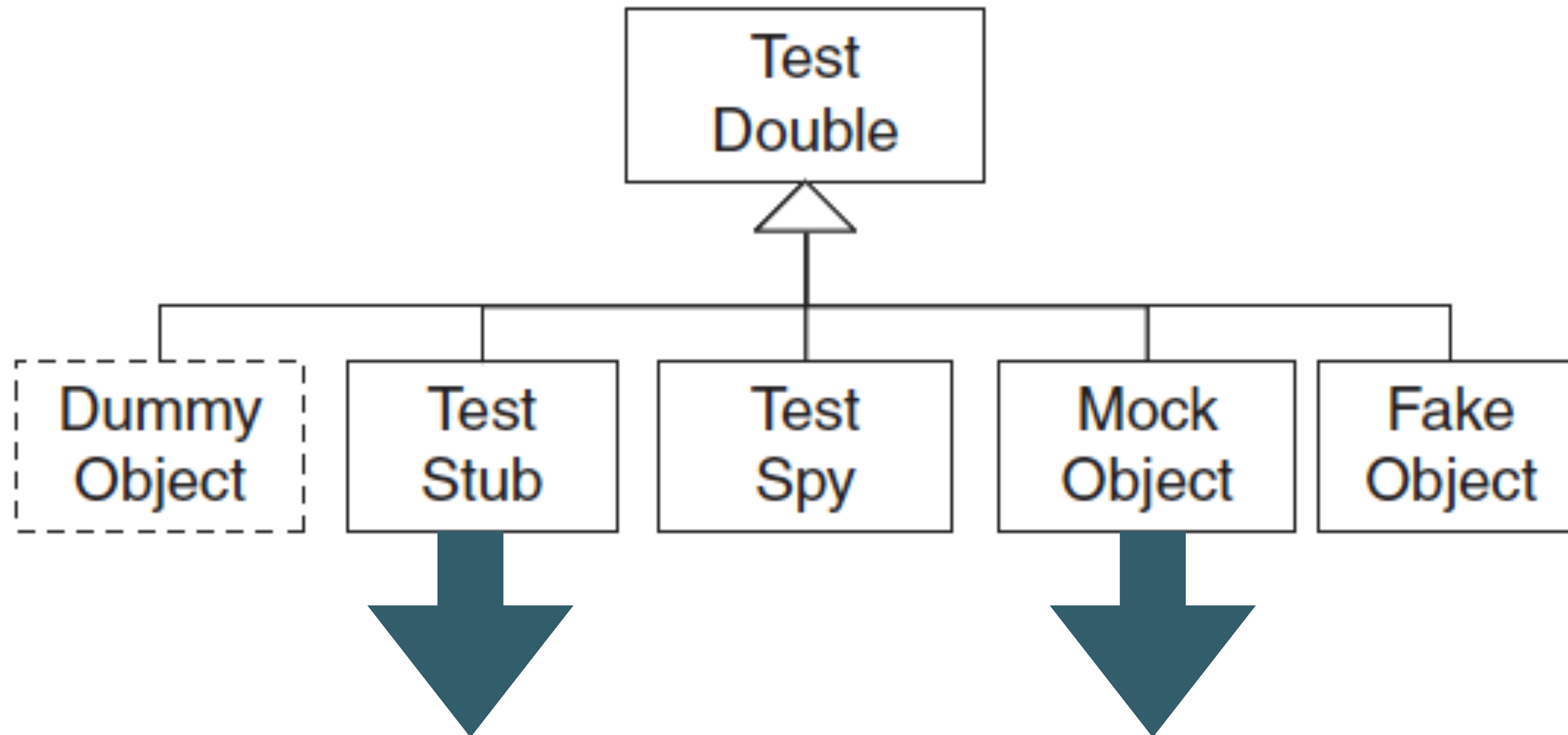


- Hard to test the SUT because it depends on other components that cannot be used in the test environment
  - Eg - those components aren't available, will not return the results needed for the test, or executing them would have undesirable side effects
- When writing a test in which we cannot use the 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

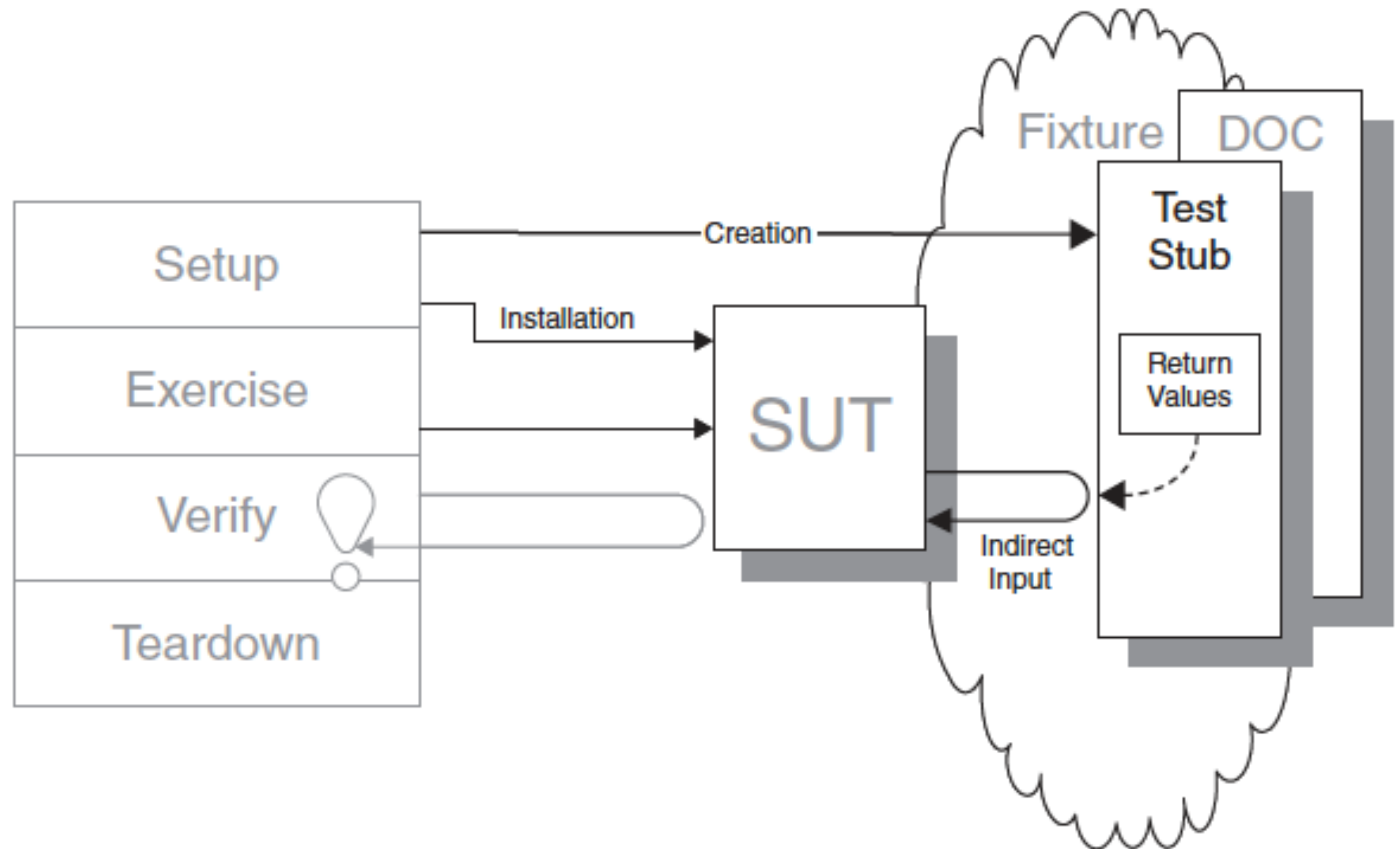


- *Test Stub:* Replace a real component on which the SUT depends so that the test has a control point for the indirect inputs of the SUT.
- *Mock Object:* an observation point to verify the indirect outputs of the SUT as it is exercised.

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);
}
```

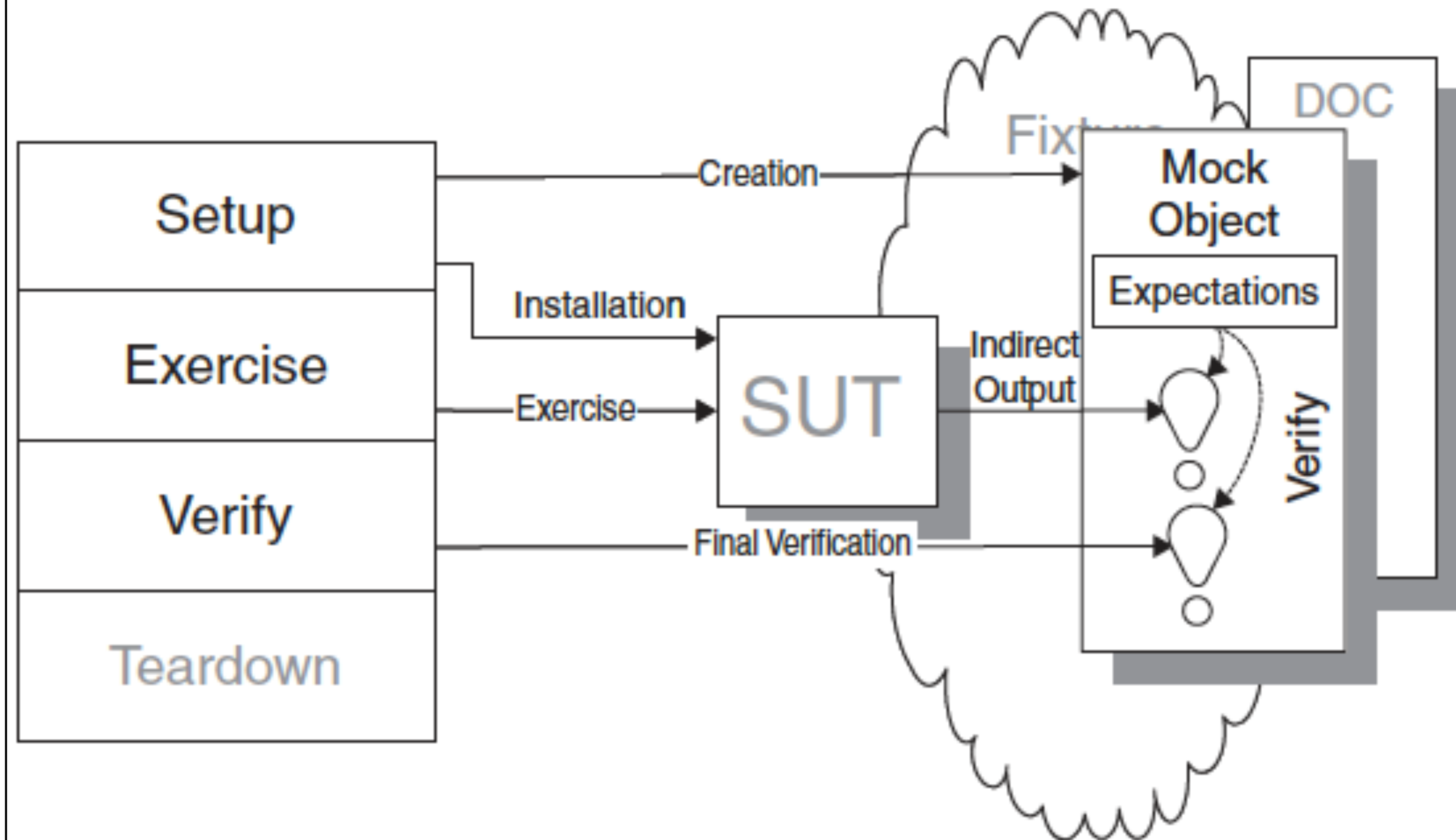
- There is no Test Stub implementation to examine for this test because the JMock framework implements the Test Stub using reflection

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 verifi cation” 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.

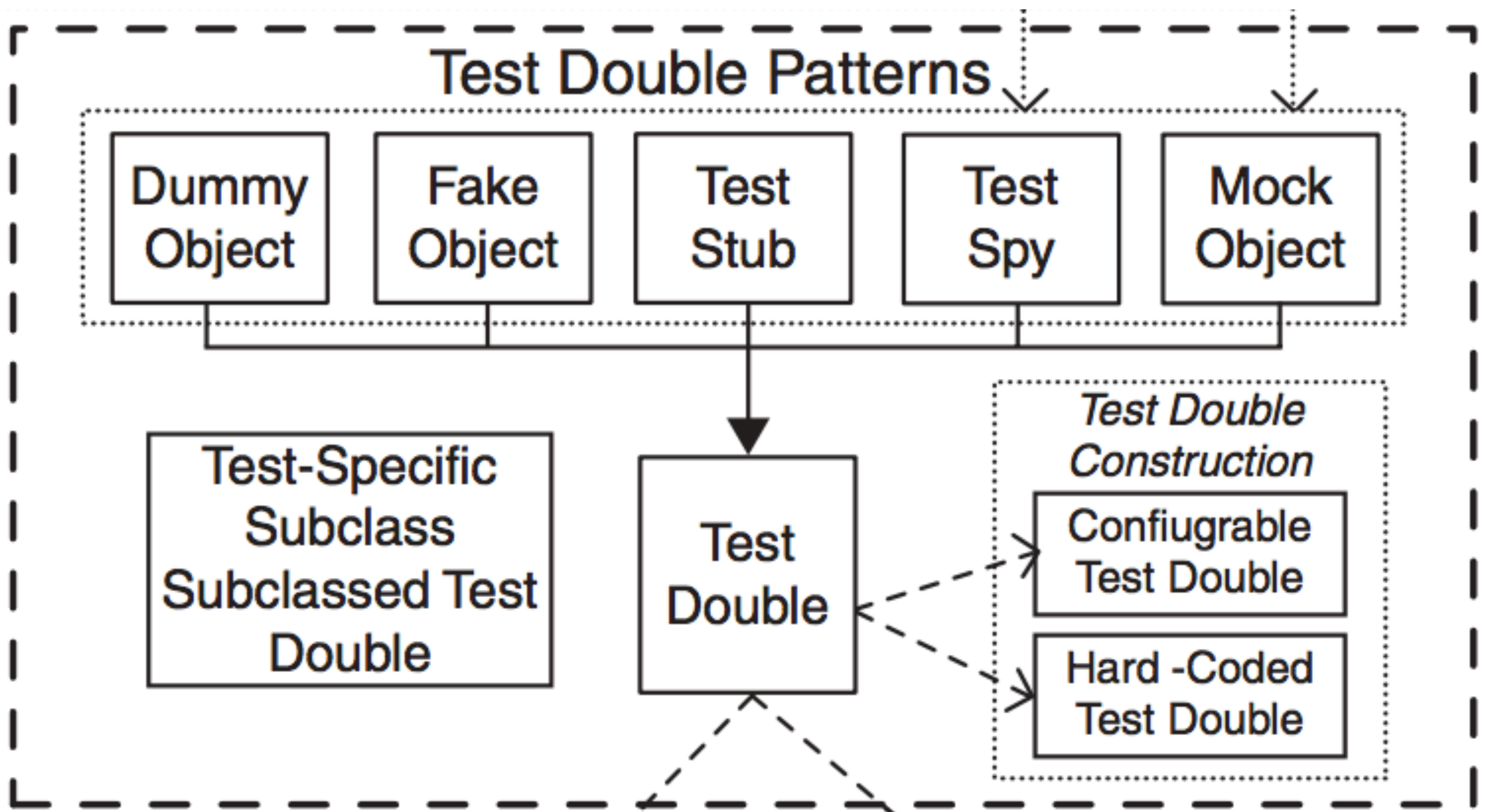
```
@Test
public void oneSubscriberReceivesAMessage()
{
  Publisher publisher = new Publisher();
  final Subscriber subscriber
    = context.mock(Subscriber.class);
  publisher.add(subscriber);

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

  publisher.publish(message);
}
```

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.


Test Double Patterns
Dummy Object
Fake Object
Test Stub
Test Spy
Mock Object
Test-Specific Subclass
Subclassed Test Double
Test Double
Test Double Construction
Confiugrable Test Double
Hard -Coded Test Double

http://www.growing-object-oriented-software.com/

"Test-Driven Development (TDD) is now an established technique for delivering better software faster. TDD is based on a simple idea: write tests for your code before you write the code itself. However, this "simple" idea takes skill and judgment to do well. Now there's a practical guide to TDD that takes you beyond the basic concepts. Drawing on a decade of experience building real-world systems, two TDD pioneers show how to let tests guide your development and "grow" software that is coherent, reliable, and maintainable."