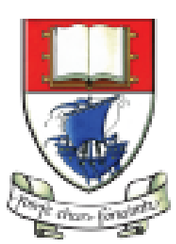
# JUnit Testing Patterns: Mocking and Doubles

Produced by:

Eamonn de Leastar (edeleastar@wit.ie)

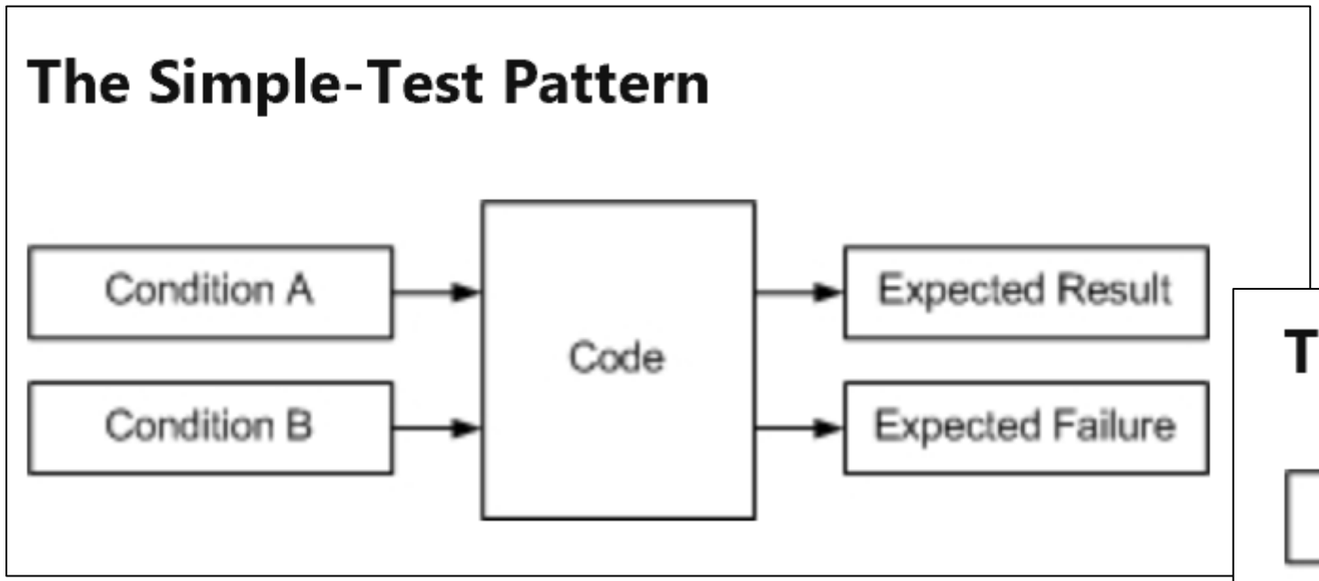
Dr. Siobhán Drohan (sdrohan@wit.ie)

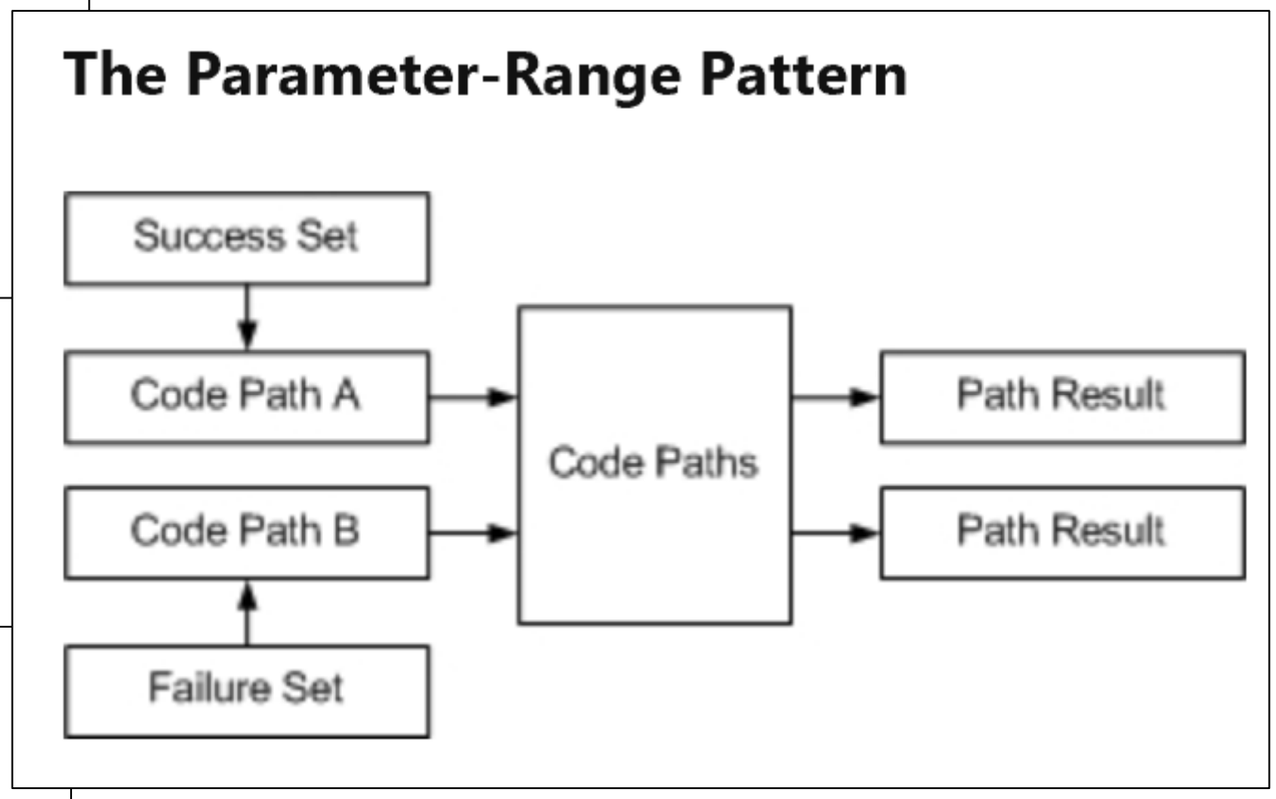


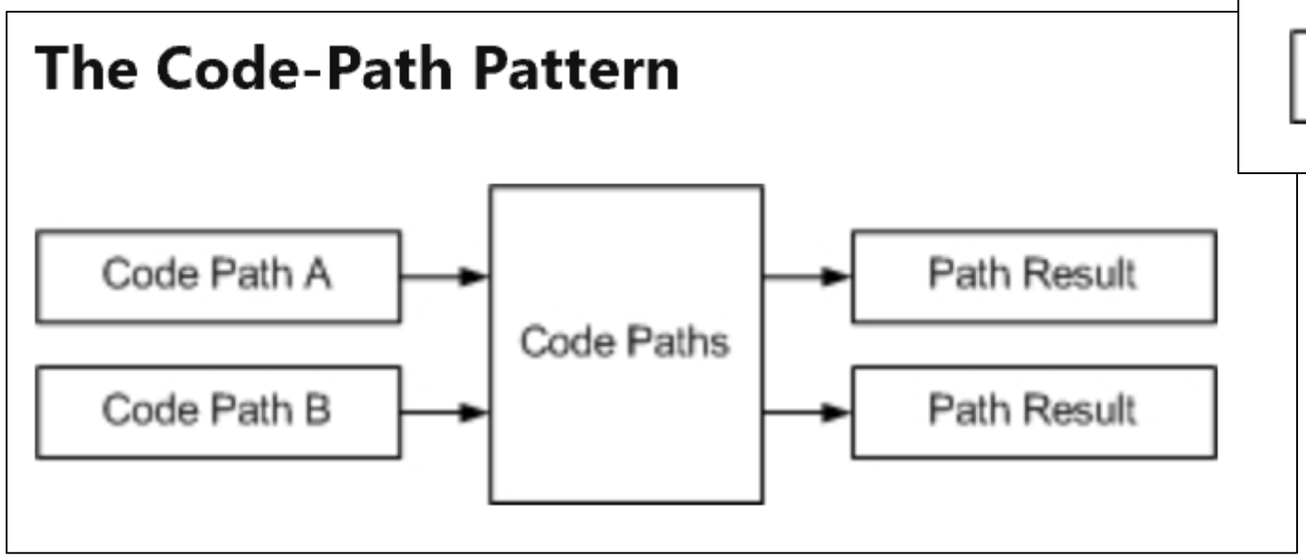
Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/

# **Unit Test Patterns:** Pass/Fail Patterns

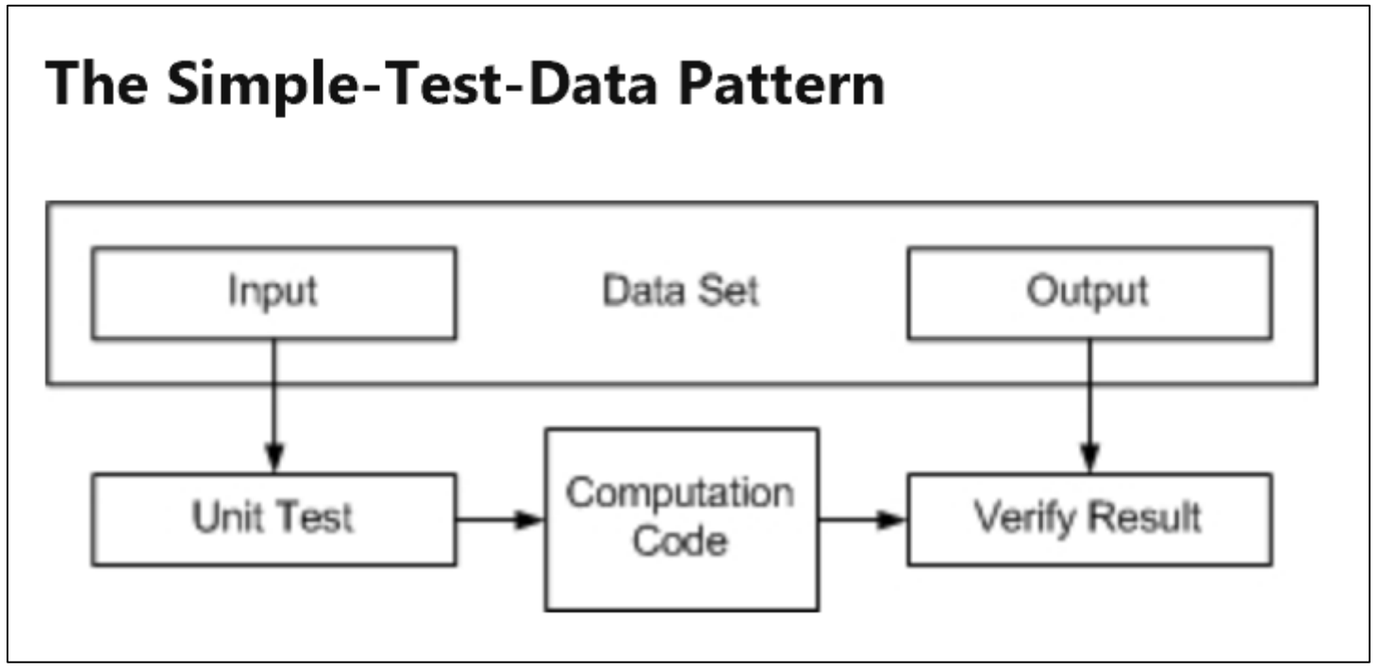


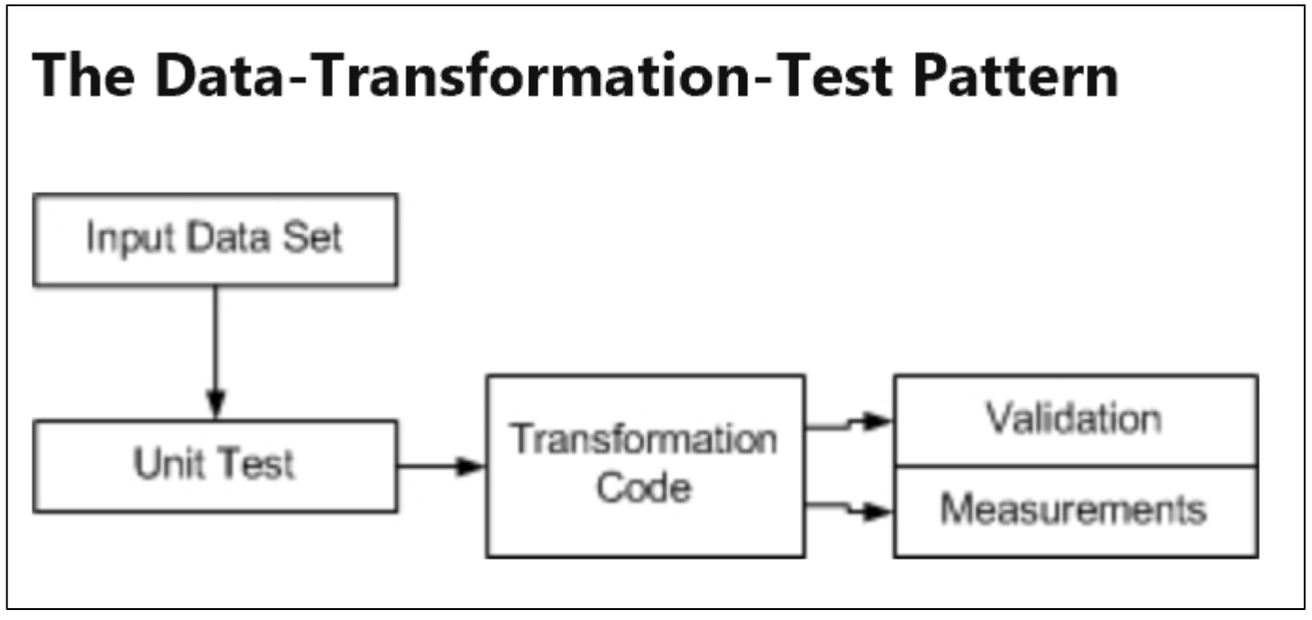




https://www.codeproject.com/articles/5772/advanced-unit-test-part-v-unit-test-patterns

# **Unit Test Patterns:** Data Driven Test Patterns

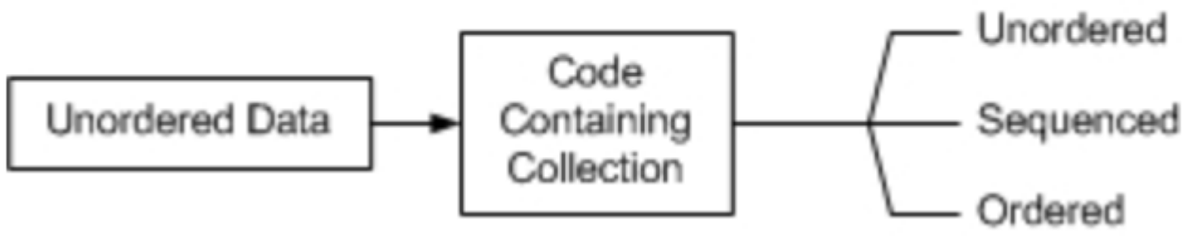


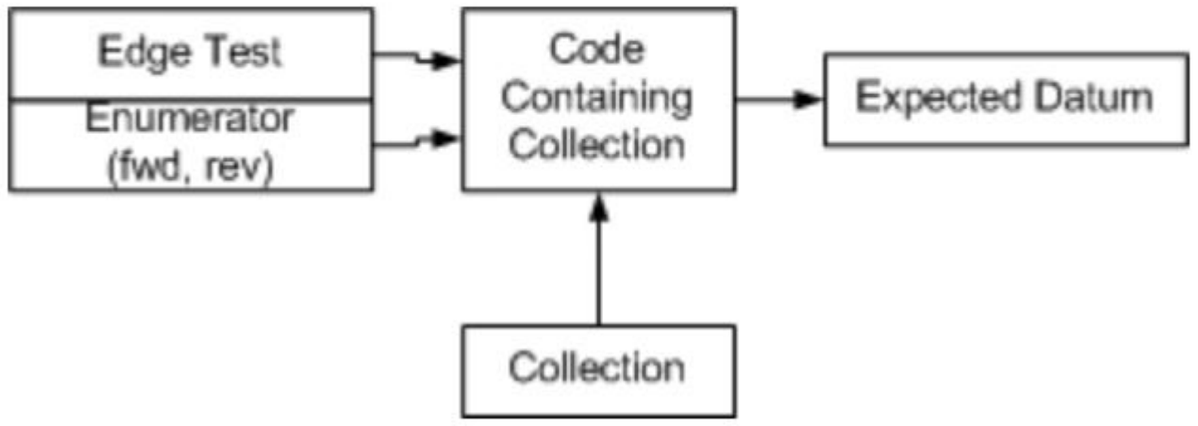

https://www.codeproject.com/articles/5772/advanced-unit-test-part-v-unit-test-patterns

# **Unit Test Patterns:** Collection Management Patterns

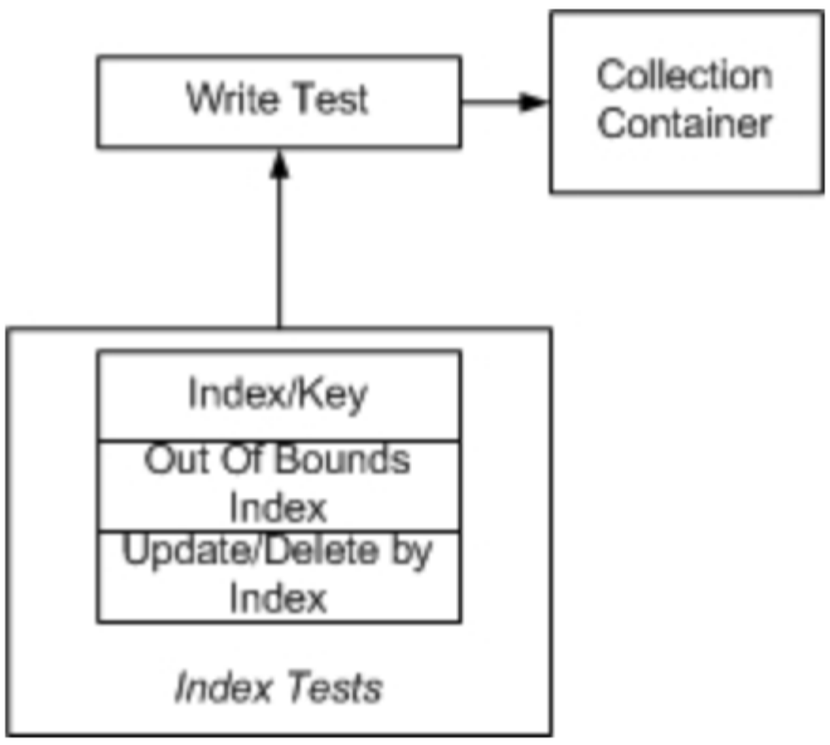
## The Collection-Order Pattern



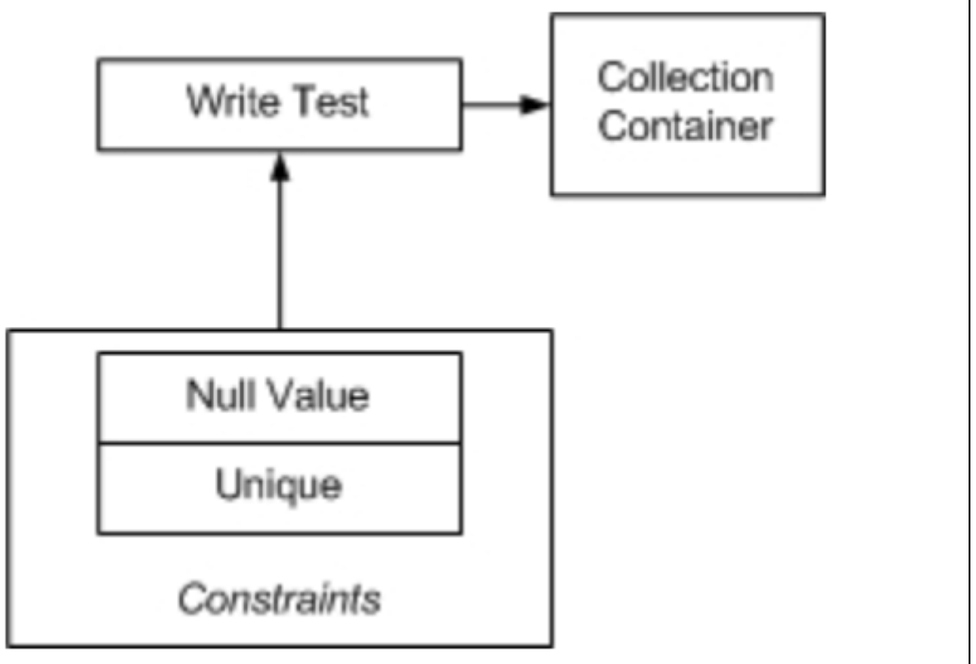

## The Enumeration Pattern




## The Collection-Indexing Pattern



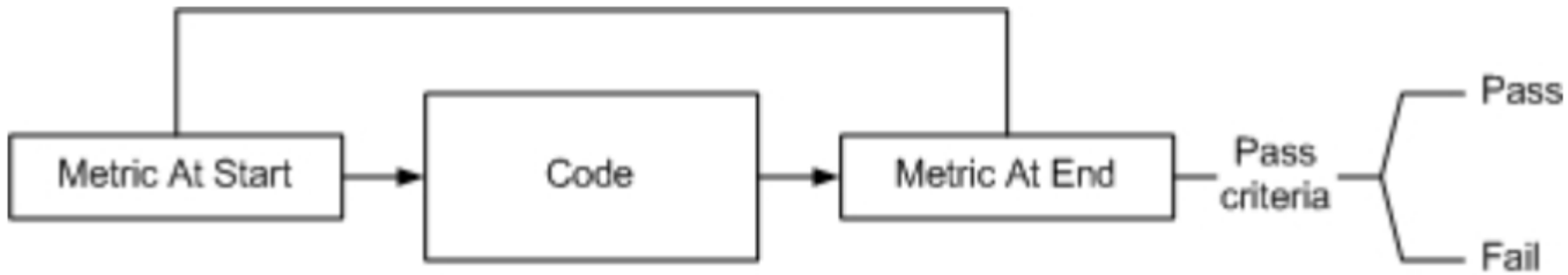

## The Collection-Constraint Pattern




https://www.codeproject.com/articles/5772/advanced-unit-test-part-v-unit-test-patterns

# **Unit Test Patterns:** Performance Management Patterns

## The Performance-Test Pattern




The basic types of performance that can be measured are:

- Memory usage (physical, cache, virtual)
- Resource (handle) utilization
- Disk utilization (physical, cache)
- Algorithm Performance (insertion, retrieval, indexing, and operation)

https://www.codeproject.com/articles/5772/advanced-unit-test-part-v-unit-test-patterns

# **Unit Test Patterns:** and many more…

- Data Transaction Patterns
- Process Patterns
- Simulation Patterns
- Multi-threading Patterns
- Stress-test Patterns
- Presentation Layer Patterns

https://www.codeproject.com/articles/5772/advanced-unit-test-part-v-unit-test-patterns

# **Unit Test Patterns:** Simulation Patterns

- Data transactions are difficult to test; they often need a pre-set configuration, an open connection, and/or an online device, etc.

- Mock objects can come to the rescue by simulating the:
    - database,
    - web service,
    - user event,
    - connection,
    - and/or hardware
- with which the code is transacting.

https://www.codeproject.com/articles/5772/advanced-unit-test-part-v-unit-test-patterns

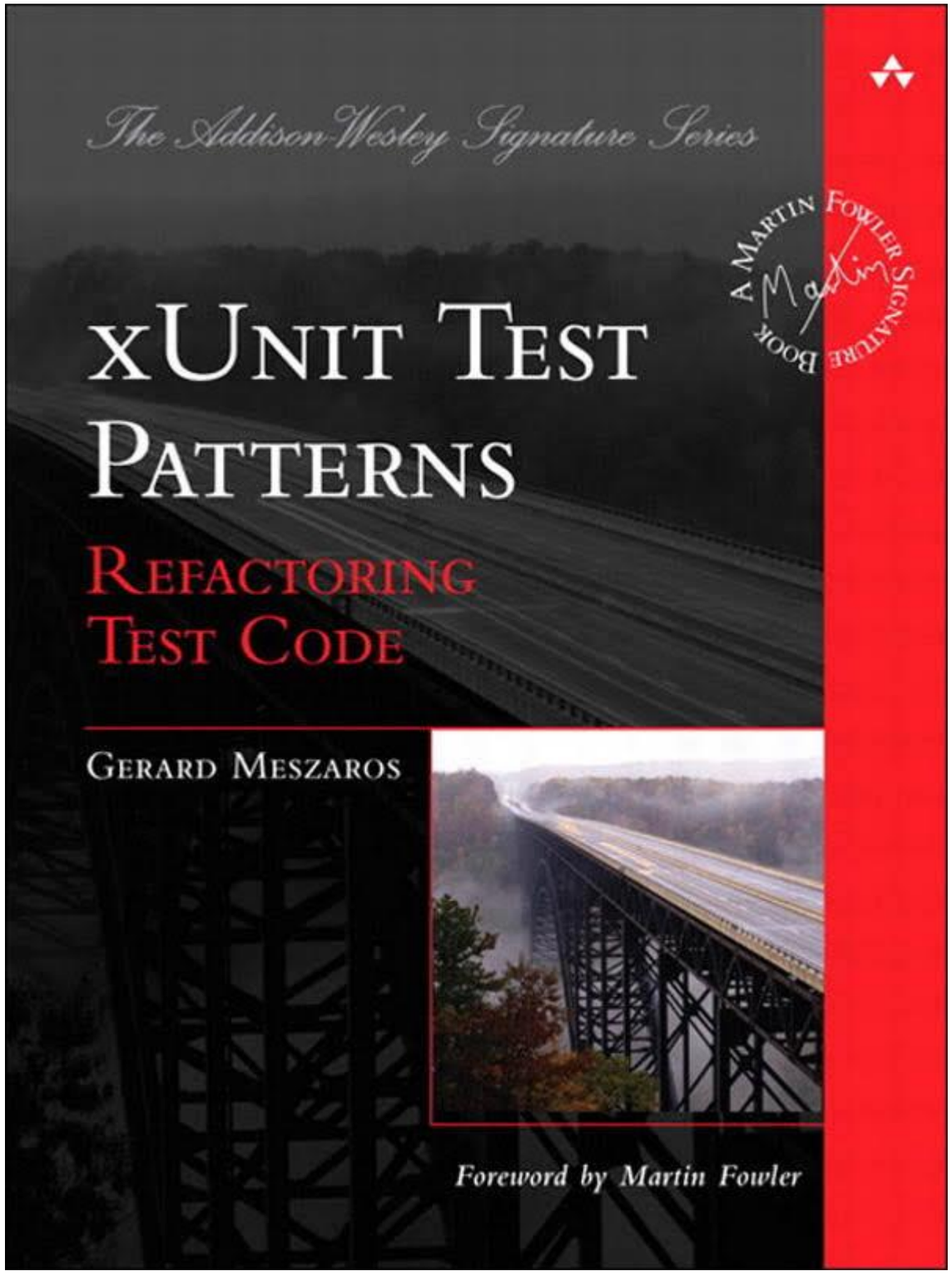
# **Unit Test Patterns:** Simulation Patterns

- Mock objects can also create failure conditions that are very difficult to reproduce in the real world:
    - a full disk,
    - a slow server,
    - a failed network hub,
    - etc.

https://www.codeproject.com/articles/5772/advanced-unit-test-part-v-unit-test-patterns

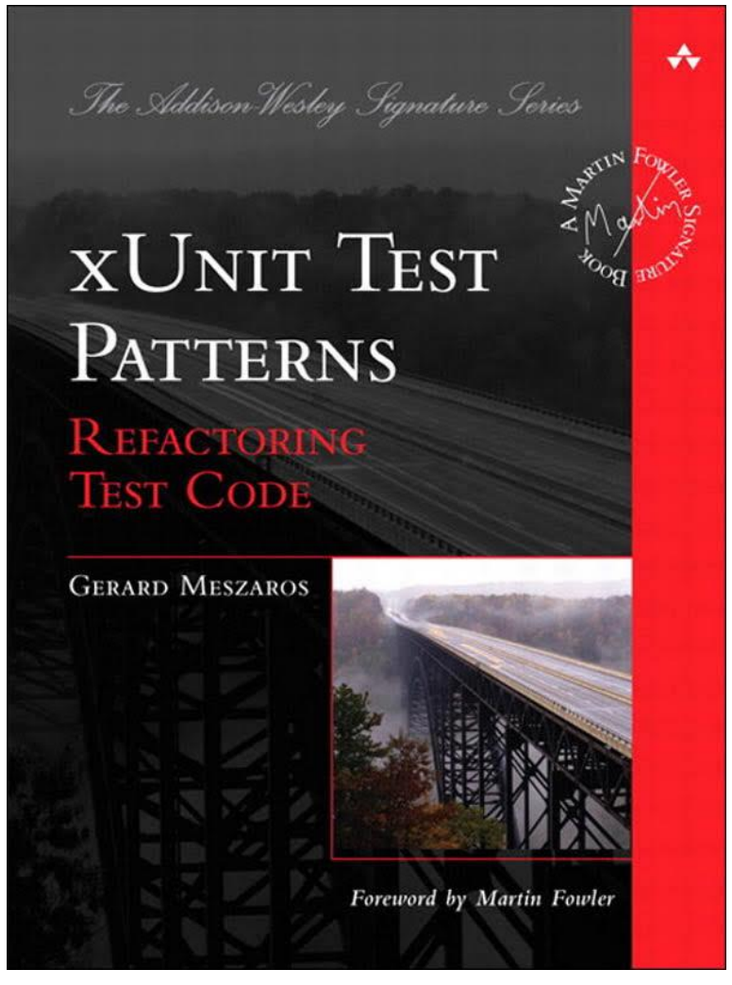
*“Automated testing is the corner stone of agile development…*
xUnit Test Patterns is the definitive guide to writing automated tests using xUnit…describes 68 proven patterns for making tests easier to write, understand, and maintain.”

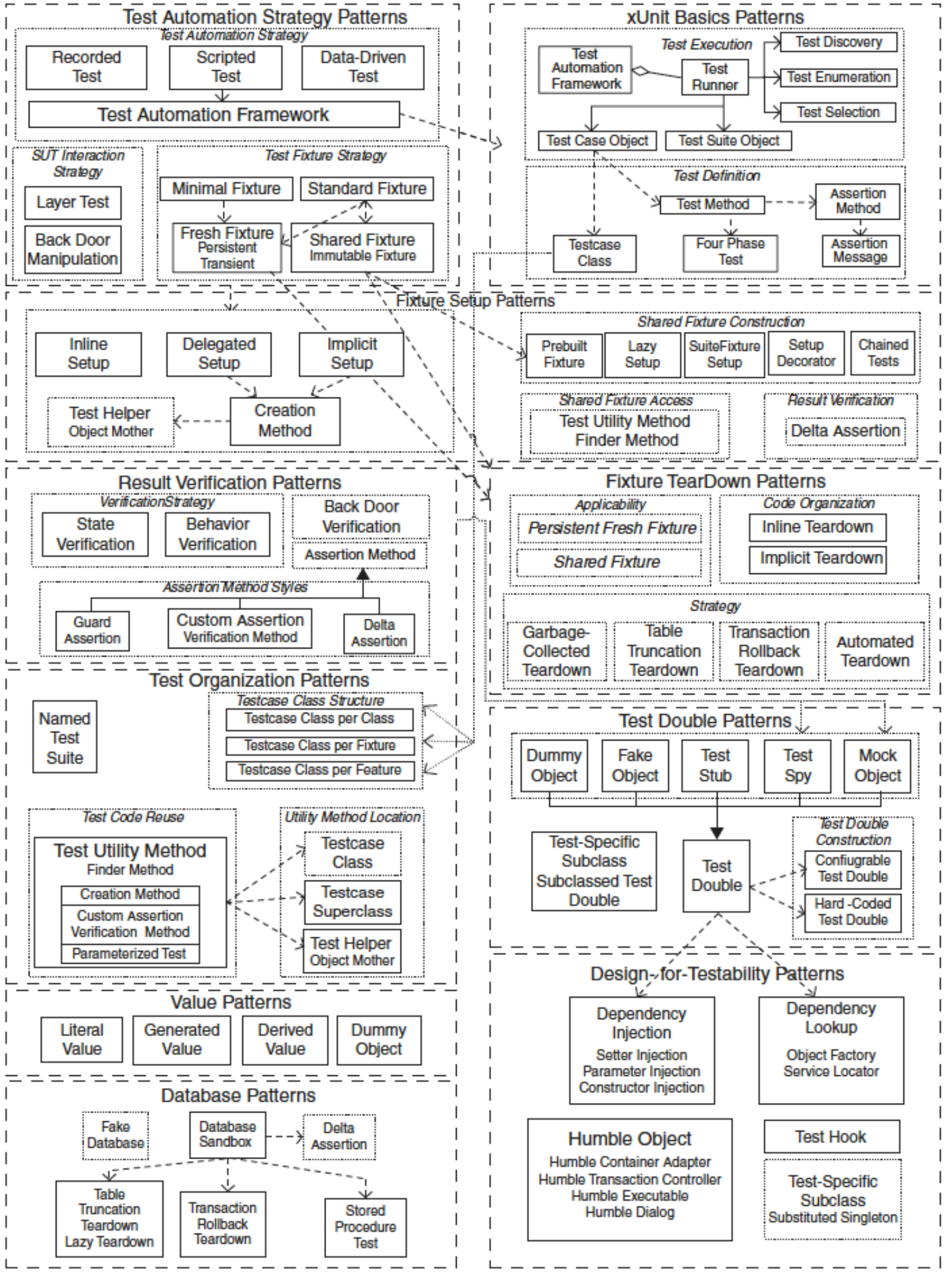
http://hillside.net/xunit-test-patterns-refactoring-test-code-hidden




- http://xunitpatterns.com/

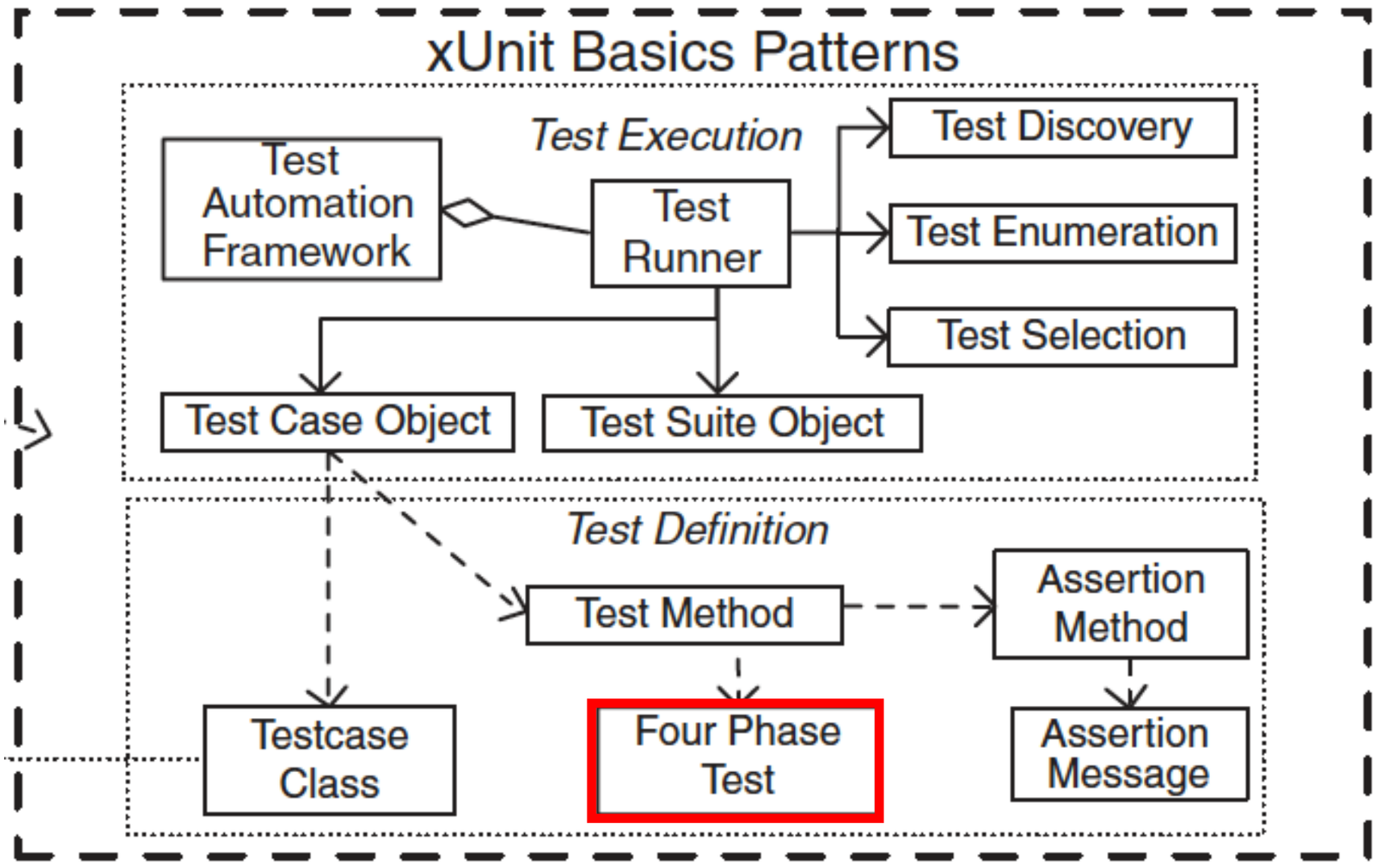
# xUnit Test Patterns



# xUnit Basics Pattern

- x indicates the programming language.
- Largely implemented in JUnit - and automatically integrated into:
  - IDEs (e.g. Eclipse)
  - Build Systems (e.g. maven)

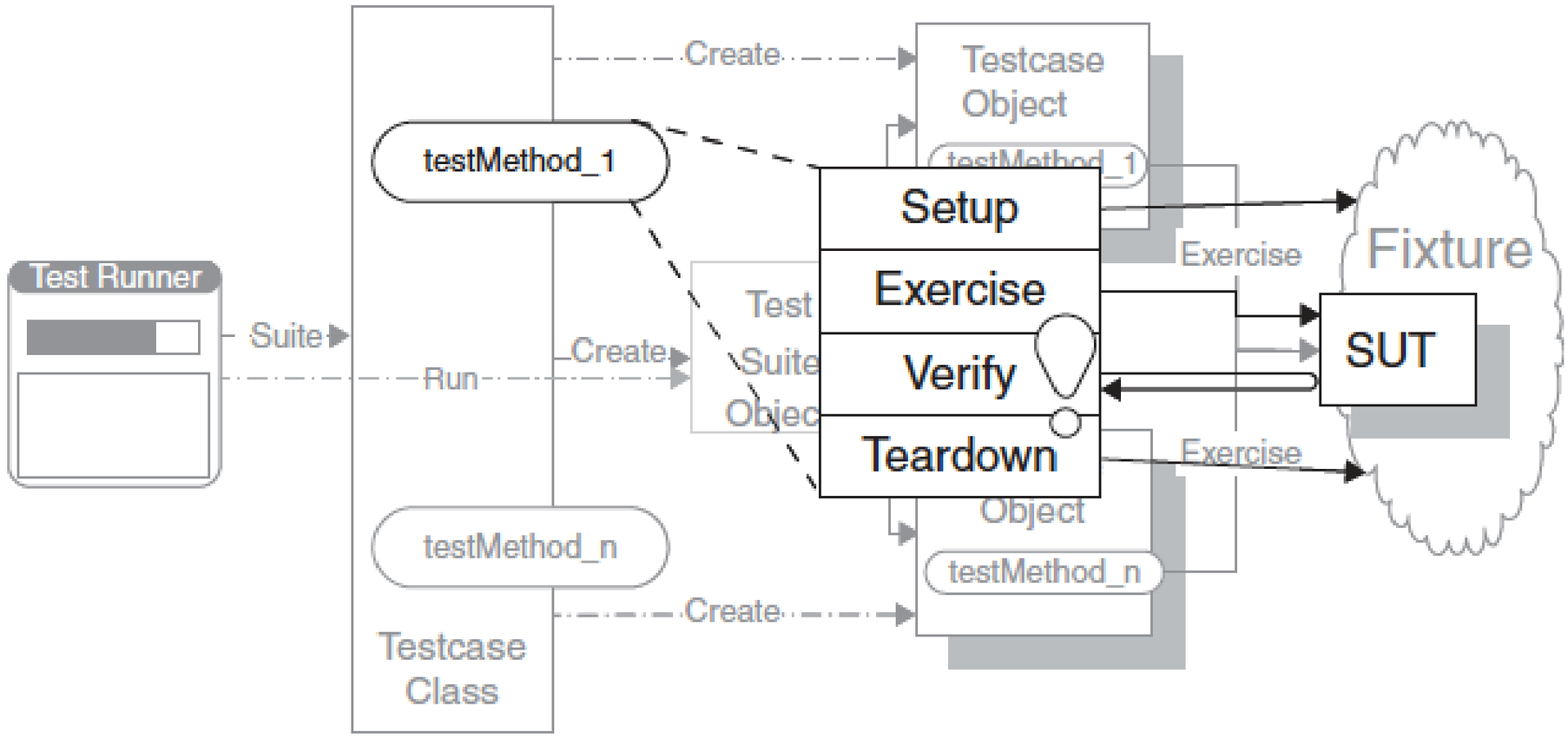


# xUnit Basics Pattern - Four Phase Test

*How do we structure our test logic to make what we are testing obvious?*

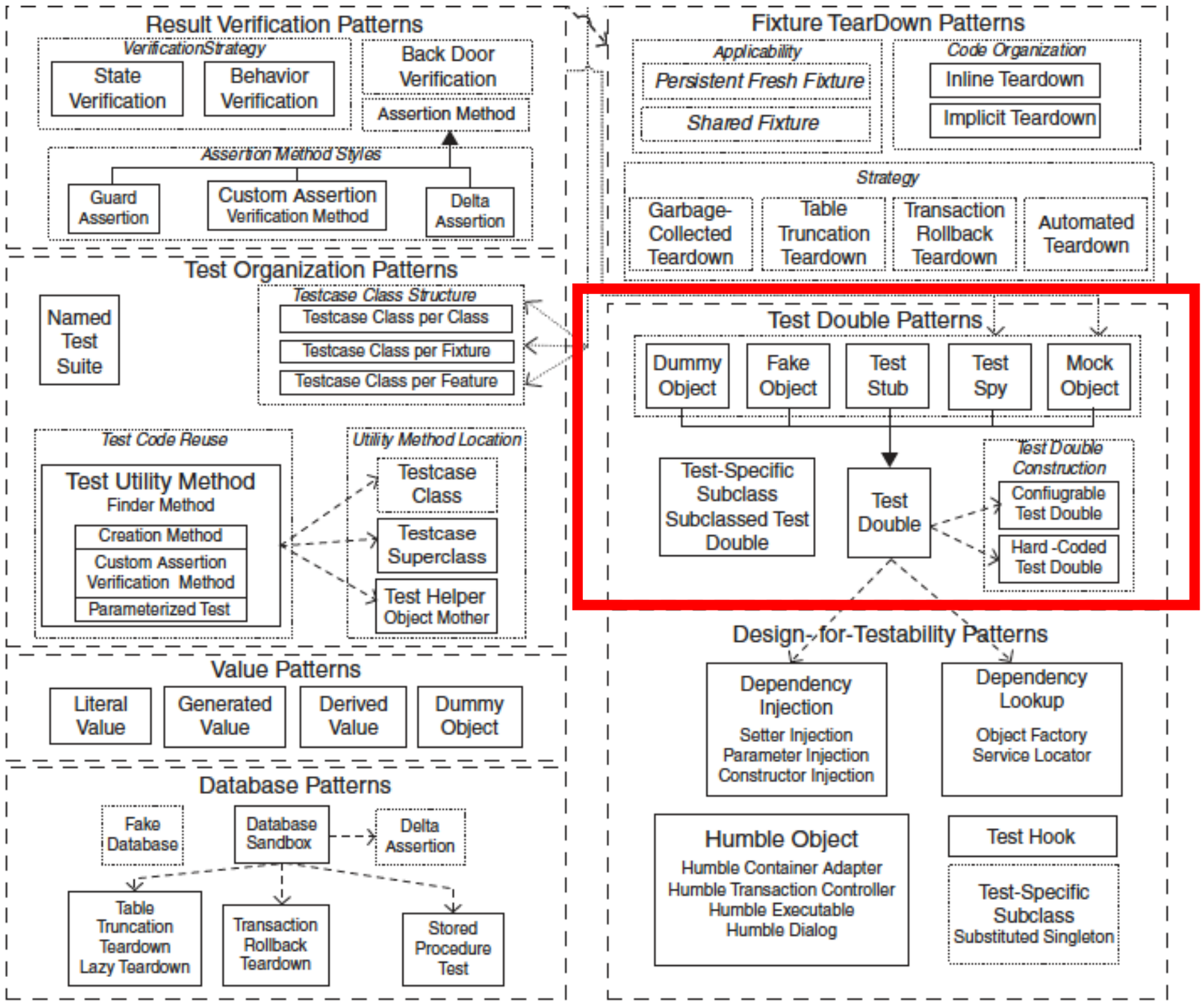
*We structure each test with four distinct parts executed in sequence.*

*Note: latest approach is AAA (Arrange, Act, Assert).*

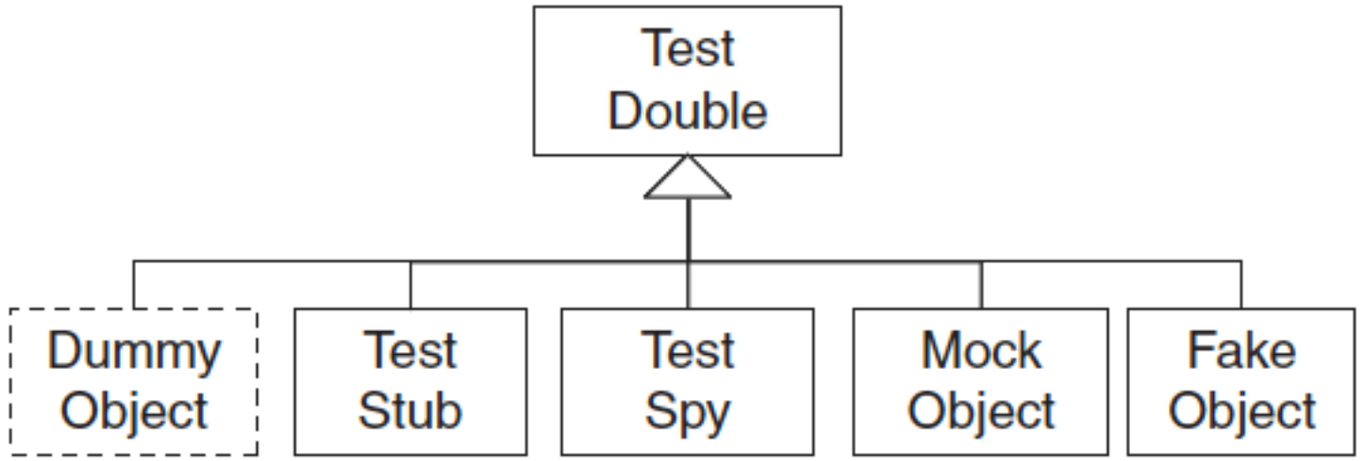



SUT = System Under Test

# xUnit Test Patterns – Test Double Patterns



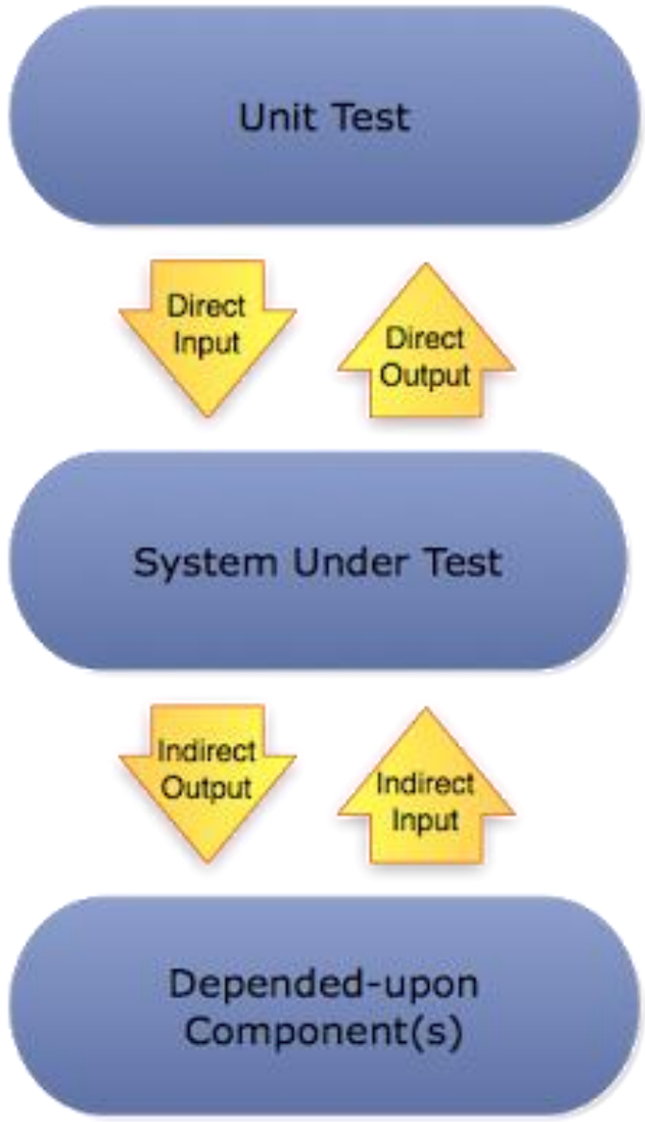
# Test Doubles – Topic List




- Understanding Test Double Terms (by example).
- Defining Test Doubles.
- Pattern Variations of Test Doubles.
- Test Stubs:
  - Hand-coded Test Stub.
  - Configurable Test Stub.
- Mock Objects:
  - Defining Mock Objects.
  - Mock Object Test Structure.
  - Maven.
  - Expectations.
  - ClassImposteriser.
- A note of caution on Test Doubles.

# Understanding Test Double Terms – by example

- Example 1:
  - We want to check that when an adder function is given a 2 and a 4 then it returns a 6.
  - In this case you're controlling what the **System Under Test** (SUT) pulls in from its environment (the 2 and the 4) and also observing what it pushes out (the 6).

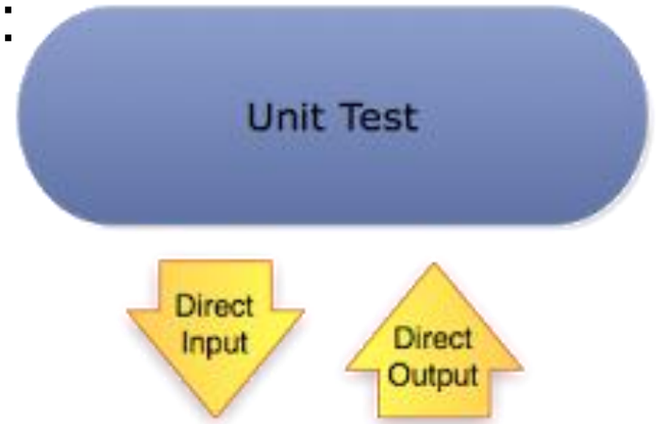



- Example 2:
  - We want to test a method that talks to a remote service and verify that if it receives an error code (pulled from the environment) when trying to read from the network then it logs the appropriate error message to a logger (SUT pushes this out).

http://blog.thepete.net/blog/2011/01/04/working-with-indirect-input-and-output/

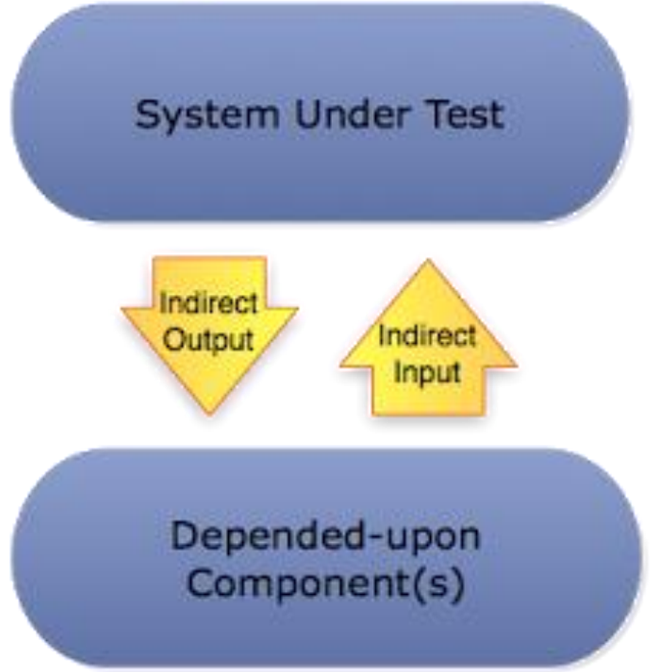
# Understanding Test Double Terms – by example

- Both examples illustrate the same fundamental practice of:
  - controlling input
  - observing output
- but they are dealing with different kinds of input and output.




- Example 1 (adder function):
  - controlling the Direct Input provided to the SUT.
  - observing the Direct Output from the SUT.

- Example 2 (remote service):
  - controlling Indirect Input from a network service (DOC).
  - observing Indirect Output to a logging service (DOC).




http://blog.thepete.net/blog/2011/01/04/working-with-indirect-input-and-output/

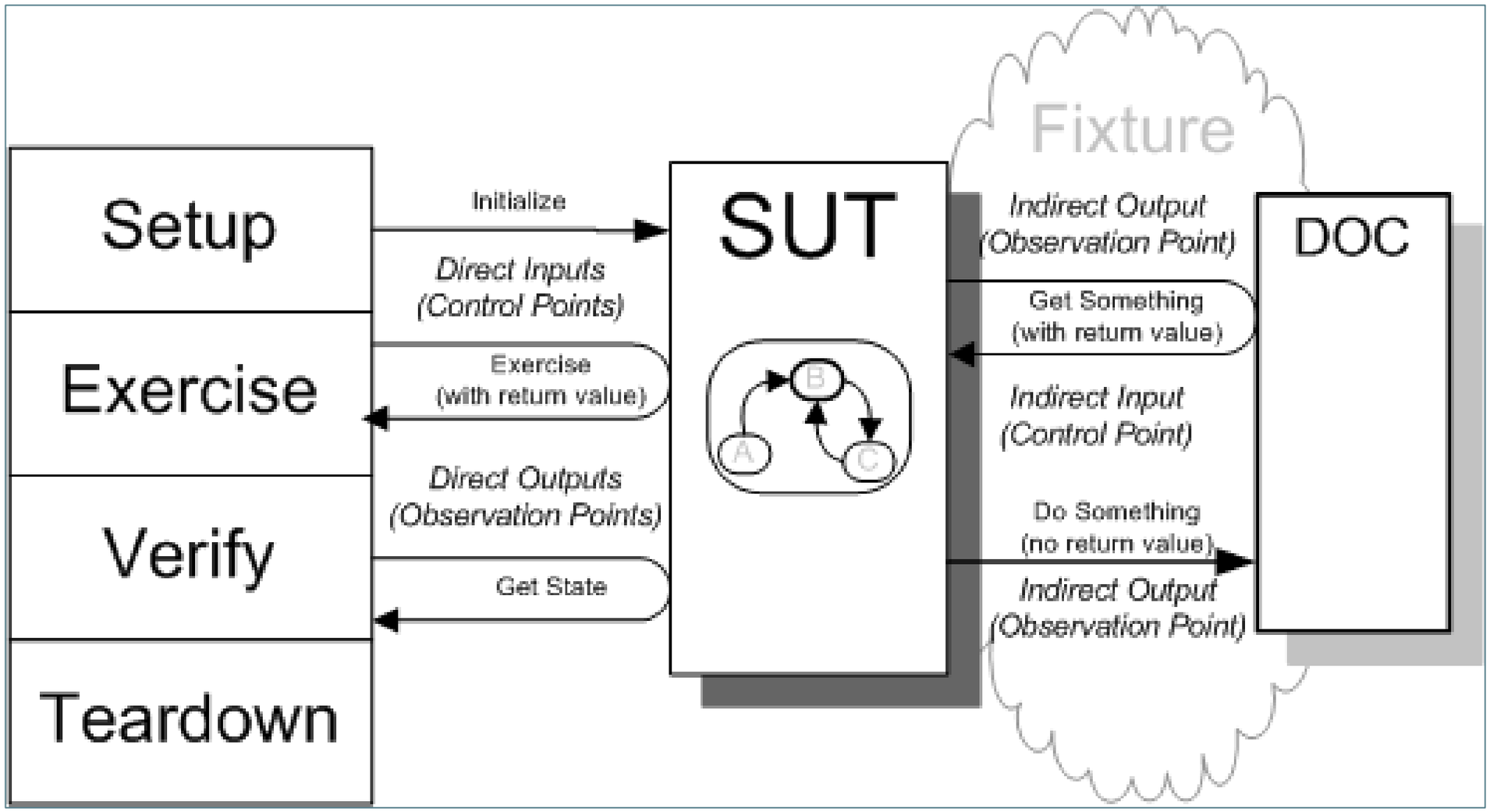
# Control and Observation Points

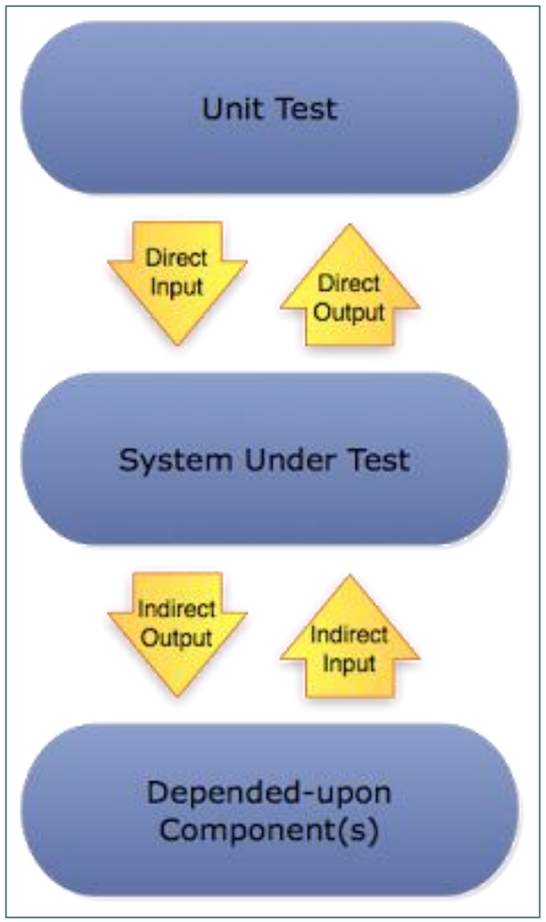
- Control Point:
  - A *control point* is how the test asks the **system under test (SUT)** to do something for it e.g. :
    - setting up or tearing down the fixture or
    - it could be used during the **exercise SUT** phase of the test.

- Observation Point:
  - A *observation point* is how the test inspects the post-exercise state of the **system under test (SUT)**.

http://xunitpatterns.com/Test%20Double.html

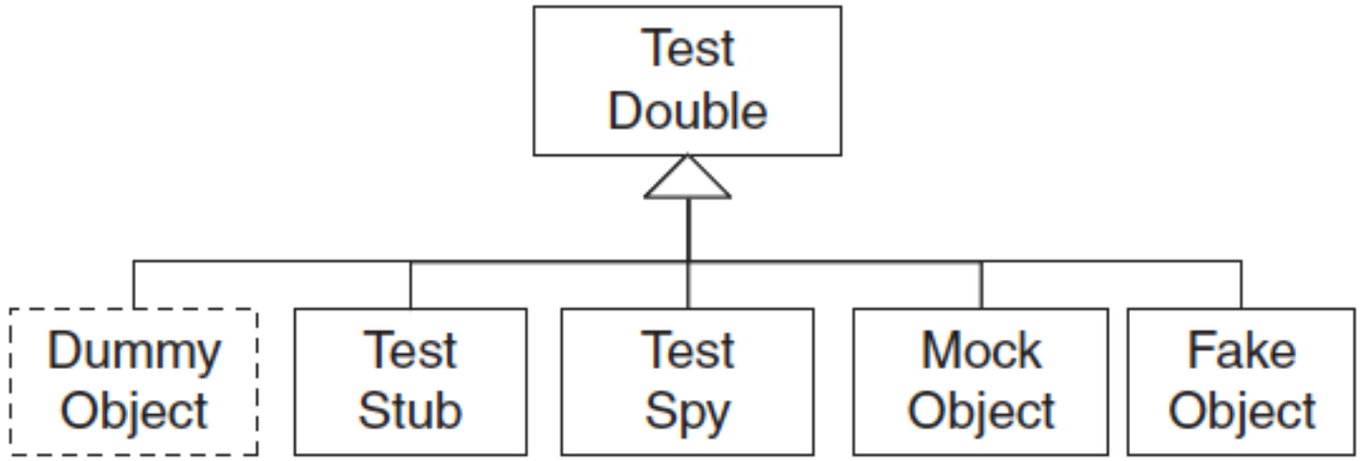
# Understanding Testing Double Terms - Diagrams




SUT - System Under Test
DOC - Depended-On Component

# Test Doubles – Topic List



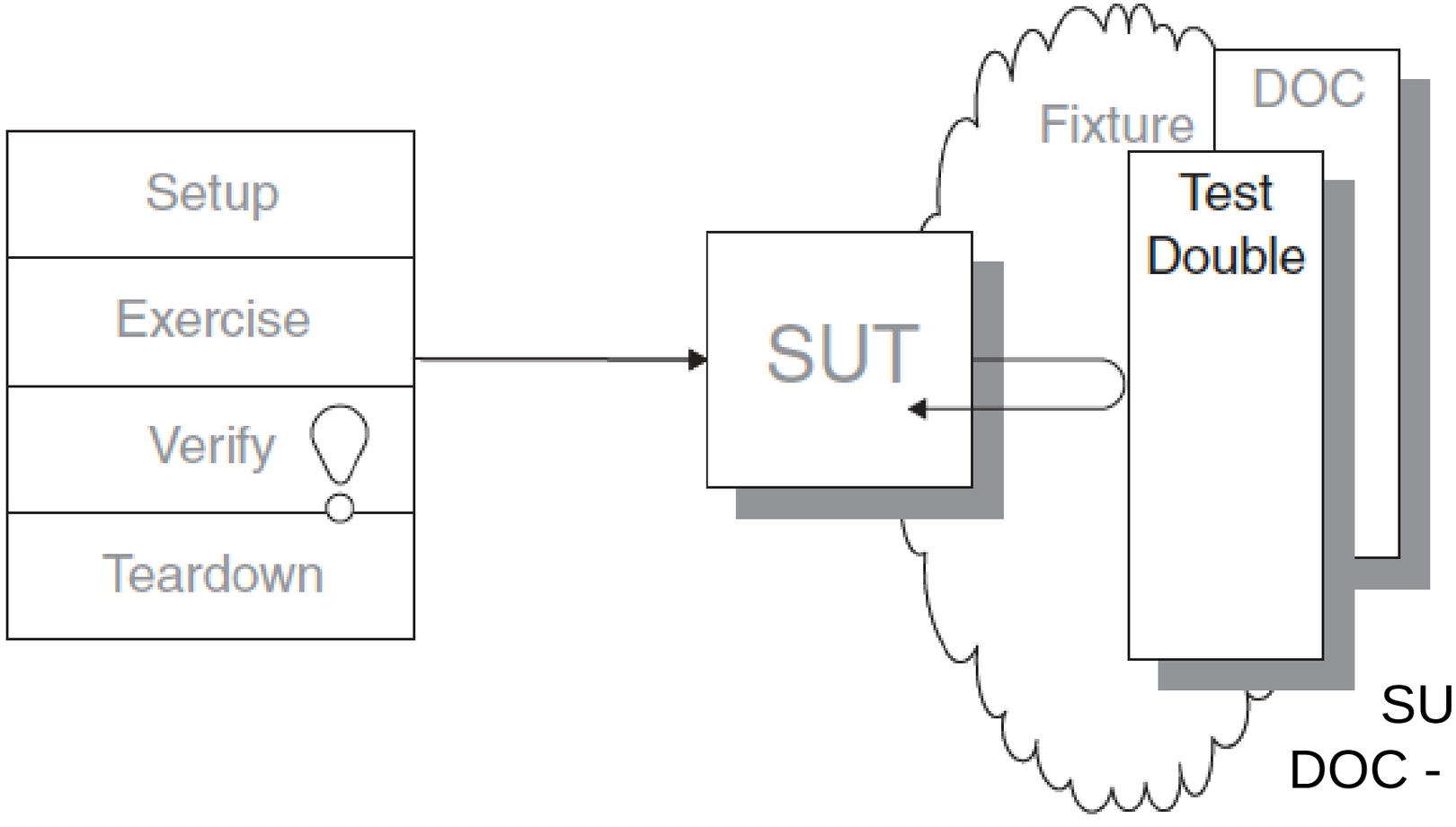

- Understanding Test Double Terms (by example).
- Defining Test Doubles.
- Pattern Variations of Test Doubles.
- Test Stubs:
  - Hand-coded Test Stub.
  - Configurable Test Stub.
- Mock Objects:
  - Defining Mock Objects.
  - Mock Object Test Structure.
  - Maven.
  - Expectations.
  - ClassImposteriser.
- A note of caution on Test Doubles.

# 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?

- It can be hard to test the SUT because it depends on other components that cannot be used in the test environment e.g.:
    - components aren’t available, or
    - components will not return the results needed for the test, or
    - executing components would have undesirable side effects, etc.

- When writing a test in which we cannot use the real Depended-on Component (DOC), we can replace it with a Test Double.

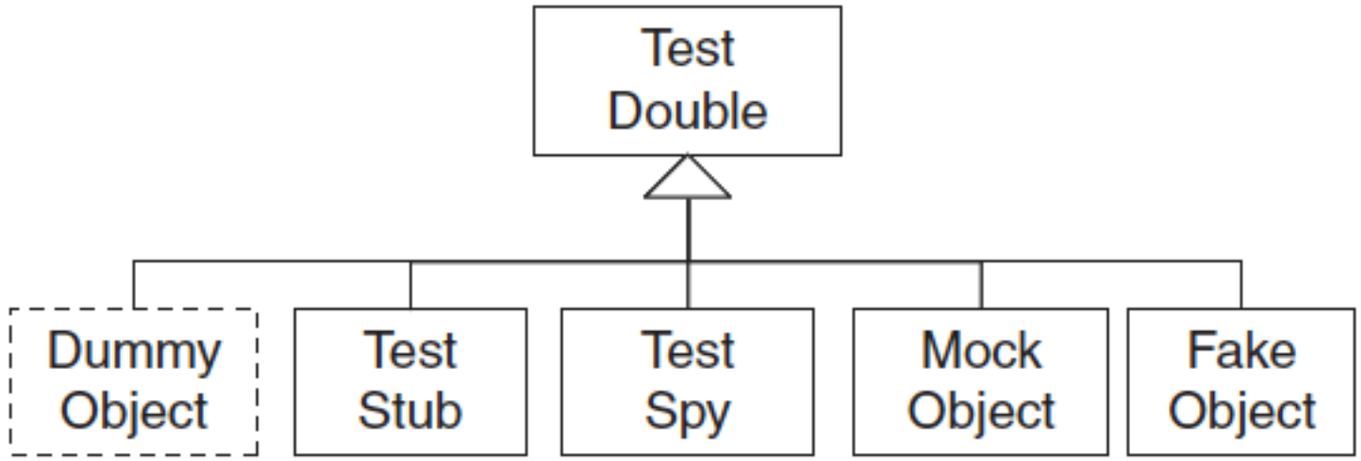
- The Test Double doesn’t have to behave exactly like DOC, it merely has to provide the same API so that the SUT thinks it is the real one.

- Called after “Stunt Double” in movie making - the stunt person takes the place of the real actor.

# When to use Test Doubles

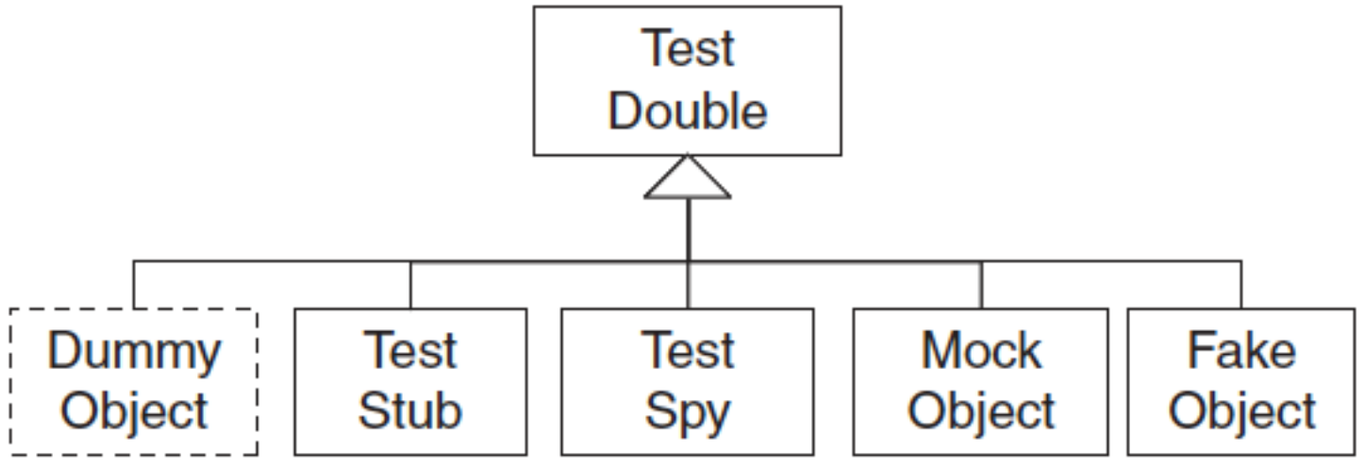
- If we have:
  - an untested requirement because neither the SUT nor its DOCs provide an observation point for the SUT's indirect output that we need to verify.
  - untested code and a DOC does not provide the control point to allow us to exercise the SUT with the necessary indirect inputs.
  - slow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

# Test Doubles – Topic List




- Understanding Test Double Terms (by example).
- Defining Test Doubles.
- Pattern Variations of Test Doubles.
- Test Stubs:
  - Hand-coded Test Stub.
  - Configurable Test Stub.
- Mock Objects:
  - Defining Mock Objects.
  - Mock Object Test Structure.
  - Maven.
  - Expectations.
  - ClassImposteriser.
- A note of caution on Test Doubles.

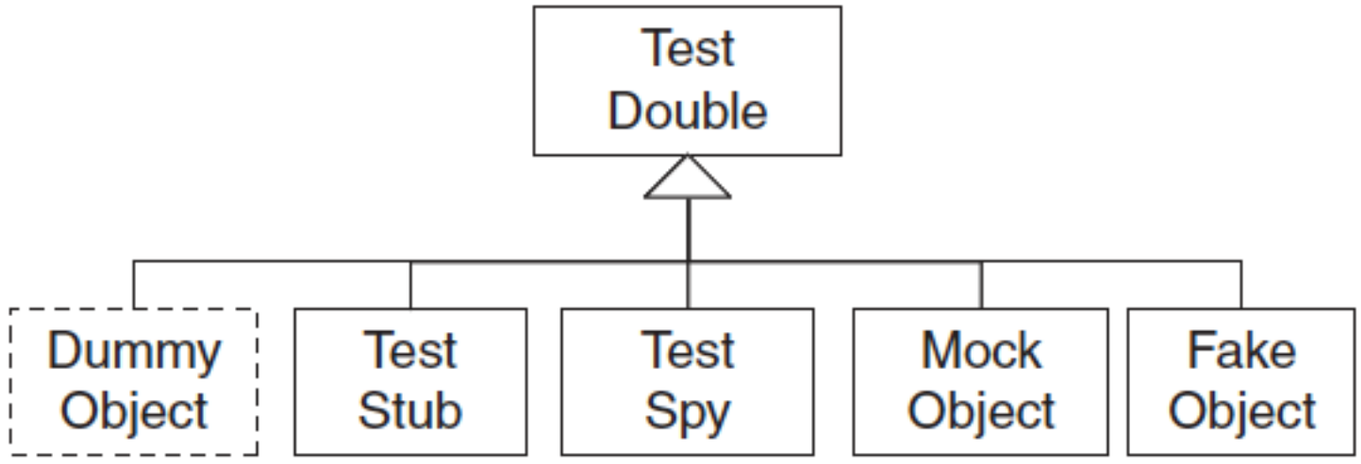
# Pattern Variations of Test Double (1)




***Dummy Object***

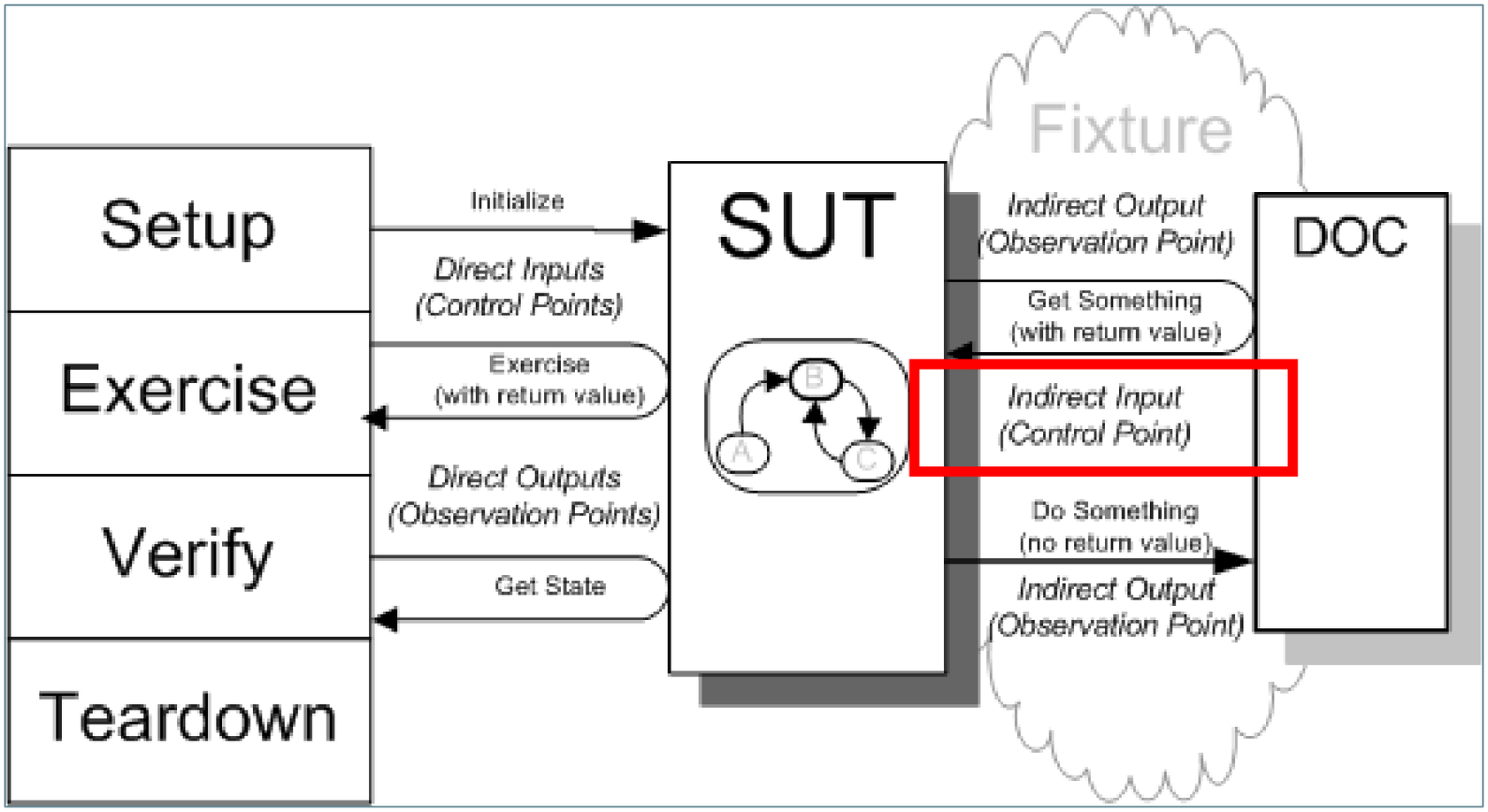
- Some method signatures of the SUT may require objects as parameters.
- If neither the test nor the SUT cares about these objects, pass in a null object reference or an instance of the Object class.
- A dummy object is passed around but never actually used; they typically just fill parameter lists.

# Pattern Variations of Test Double (2)

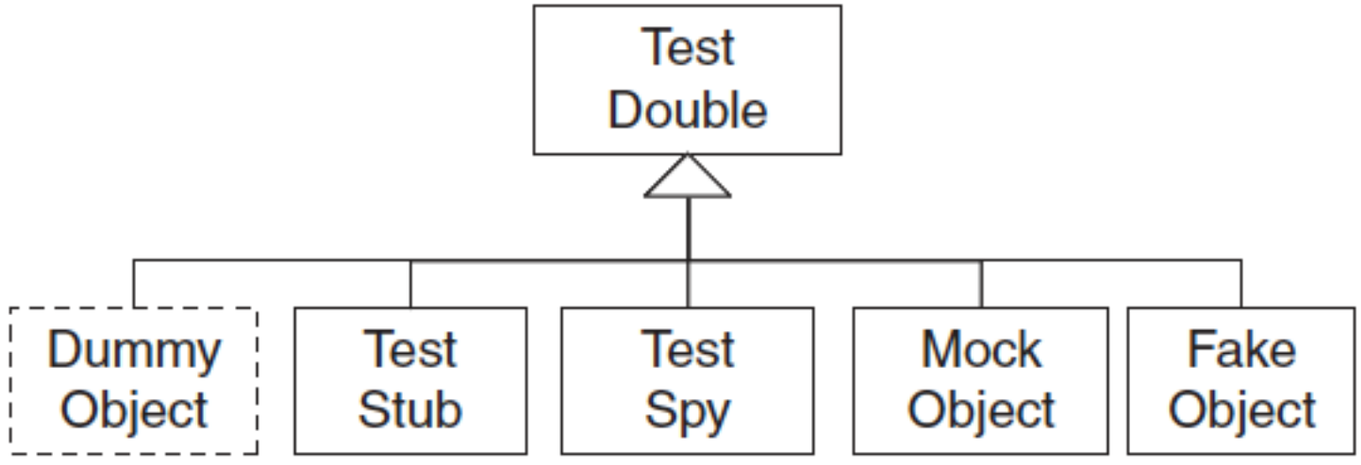



***Test Stub***

- Replace a real DOC on which the SUT depends so that the test has a control point for the indirect inputs of the SUT.
- Provides canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.

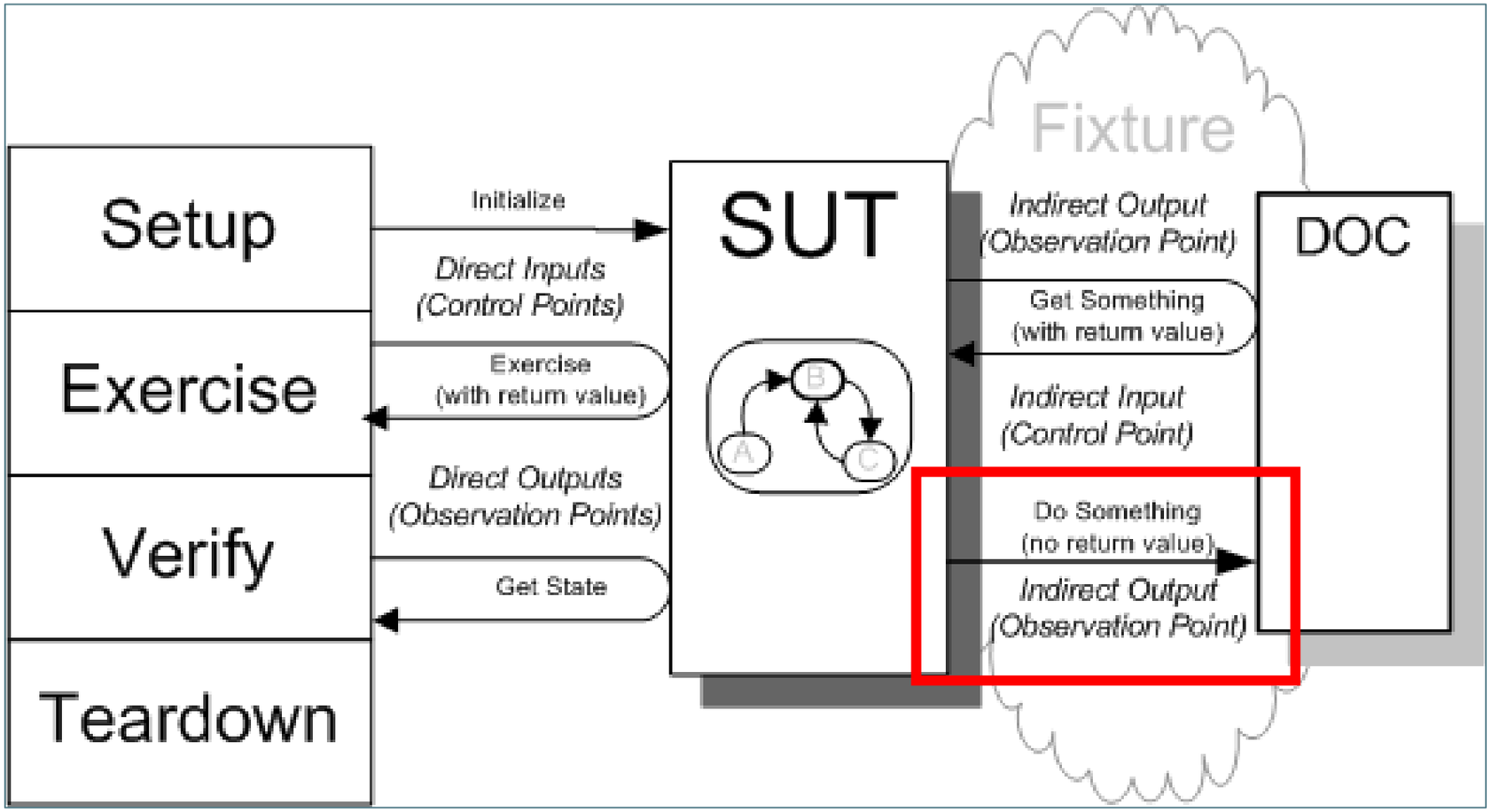


# Pattern Variations of Test Double (3)

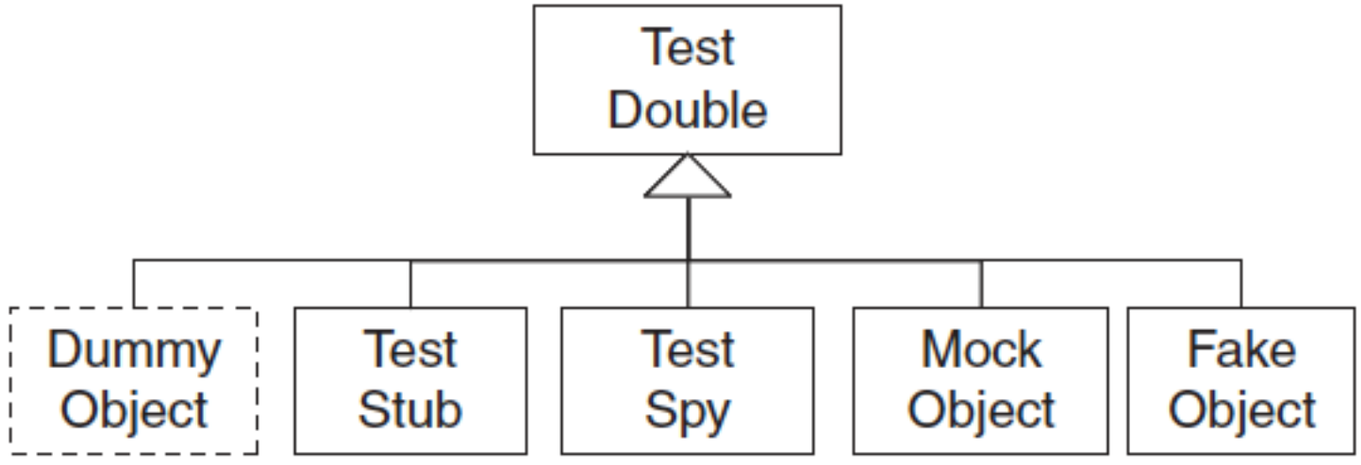



## *Test Spy*

- A more capable version of a Test Stub.
- Implements an observation point that exposes the indirect outputs of the SUT so they can be verified.

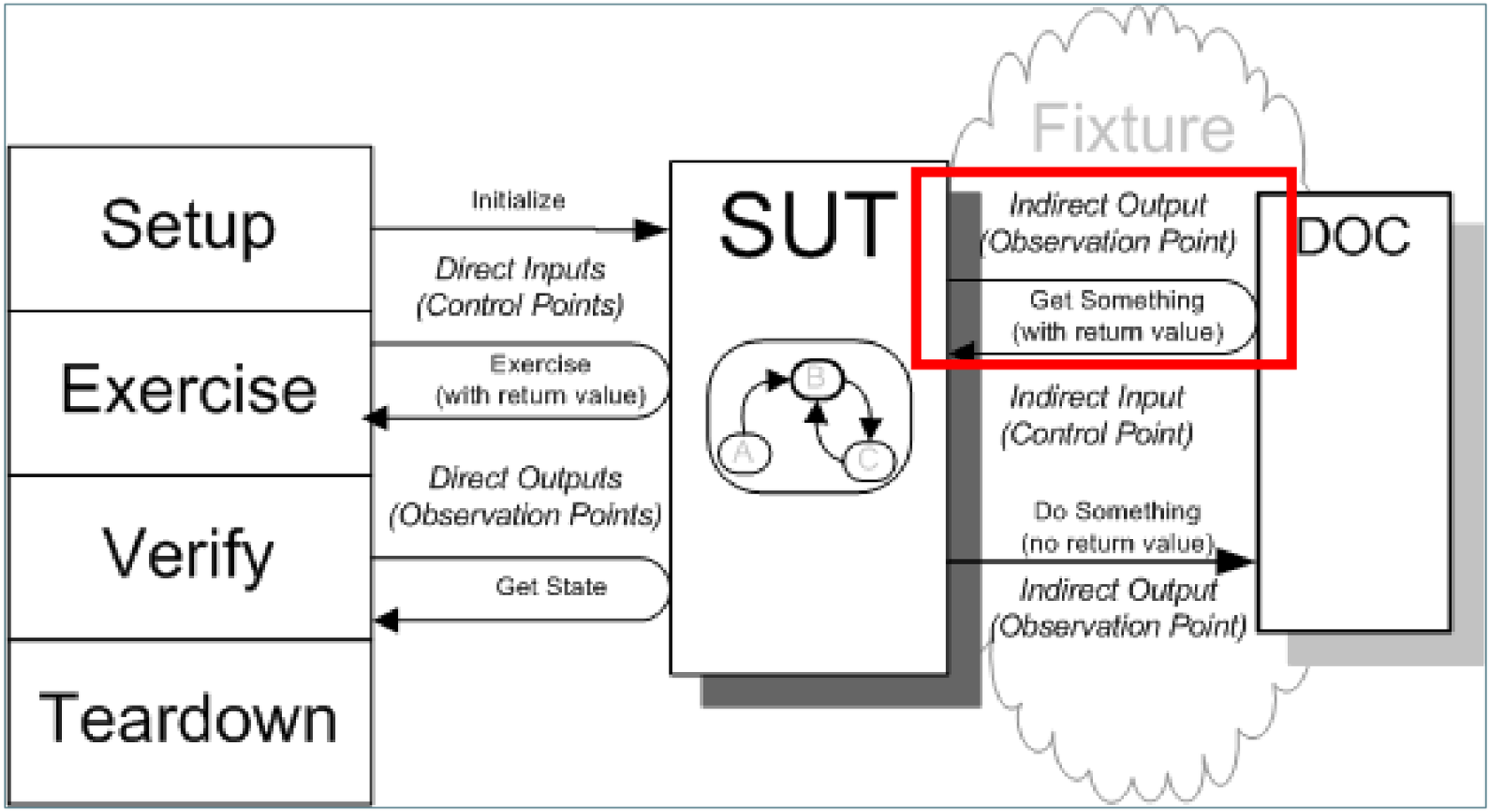


# Pattern Variations of Test Double (4)

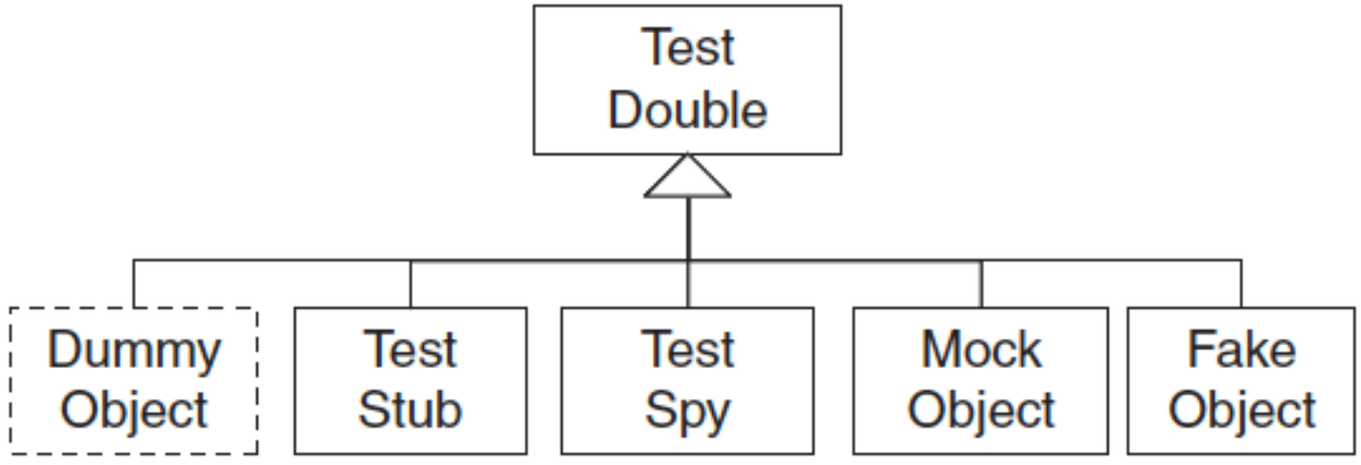



***Mock Object:***

- An observation point to verify the indirect outputs of the SUT as it is exercised.
- They are objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

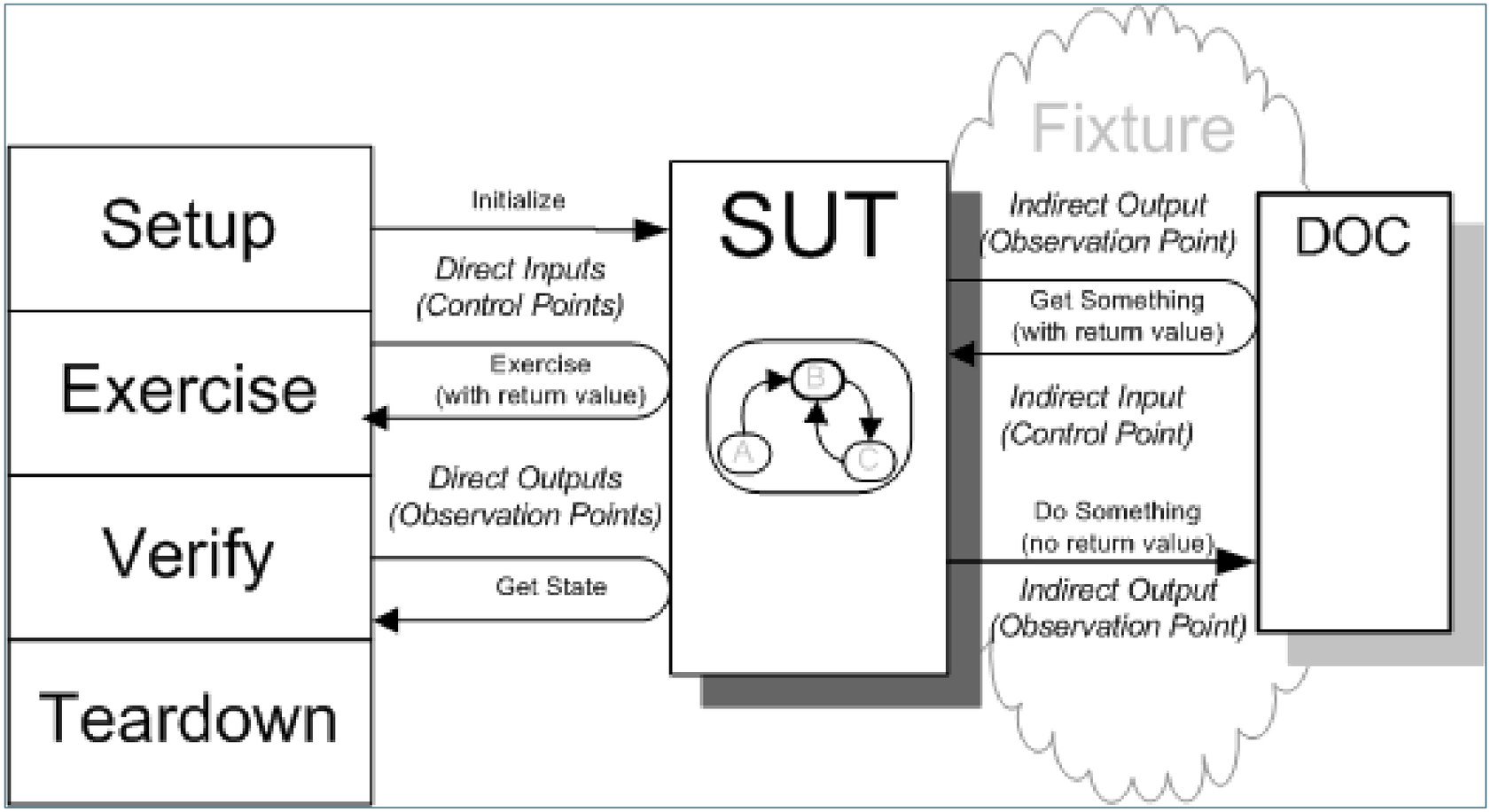


# Pattern Variations of Test Double (5)

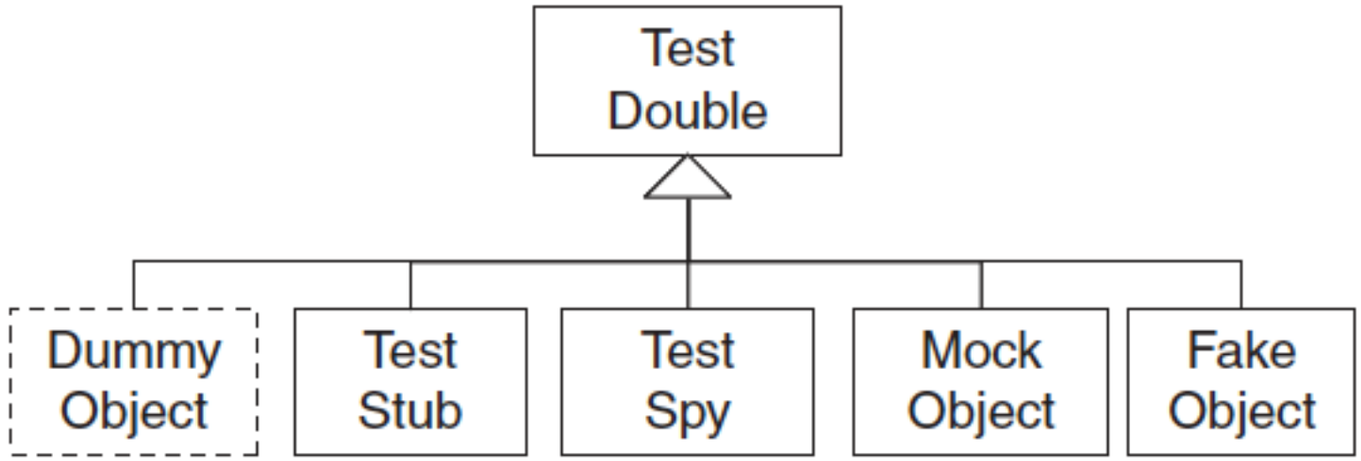



***Fake Object*:**

- Have real working implementations, but usually takes some shortcuts.
- 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 Doubles – Topic List



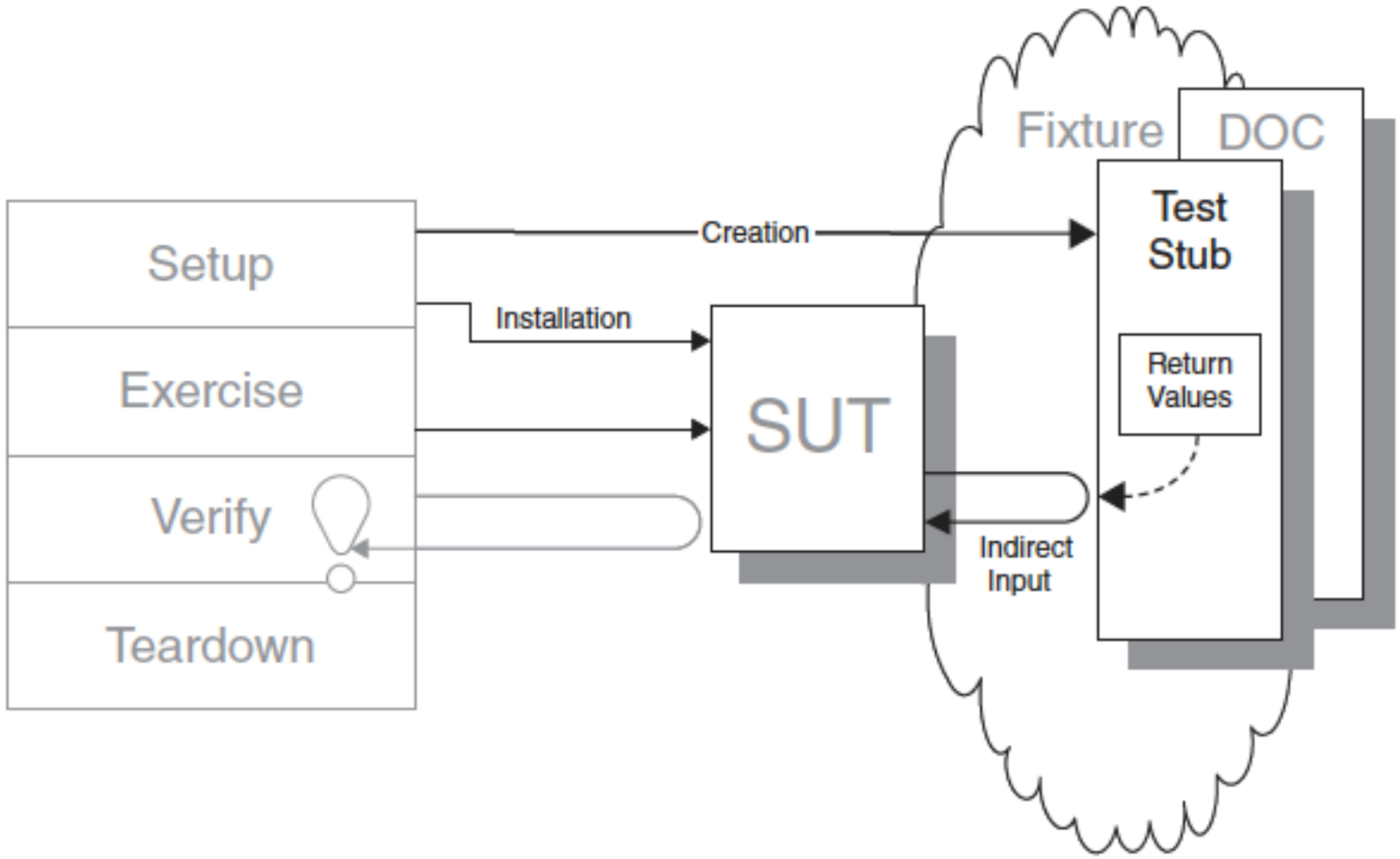

- Understanding Test Double Terms (by example).
- Defining Test Doubles.
- Pattern Variations of Test Doubles.
- Test Stubs:
  - Hand-coded Test Stub.
  - Configurable Test Stub.
- Mock Objects:
  - Defining Mock Objects.
  - Mock Object Test Structure.
  - Maven.
  - Expectations.
  - ClassImposteriser.
- A note of caution on Test Doubles.

http://xunitpatterns.com/Test%20Stub.html

# Test Stub

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

We replace a real object (DOC) with a test-specific object (Test Stub) that feeds the desired indirect inputs into the system under test (SUT).



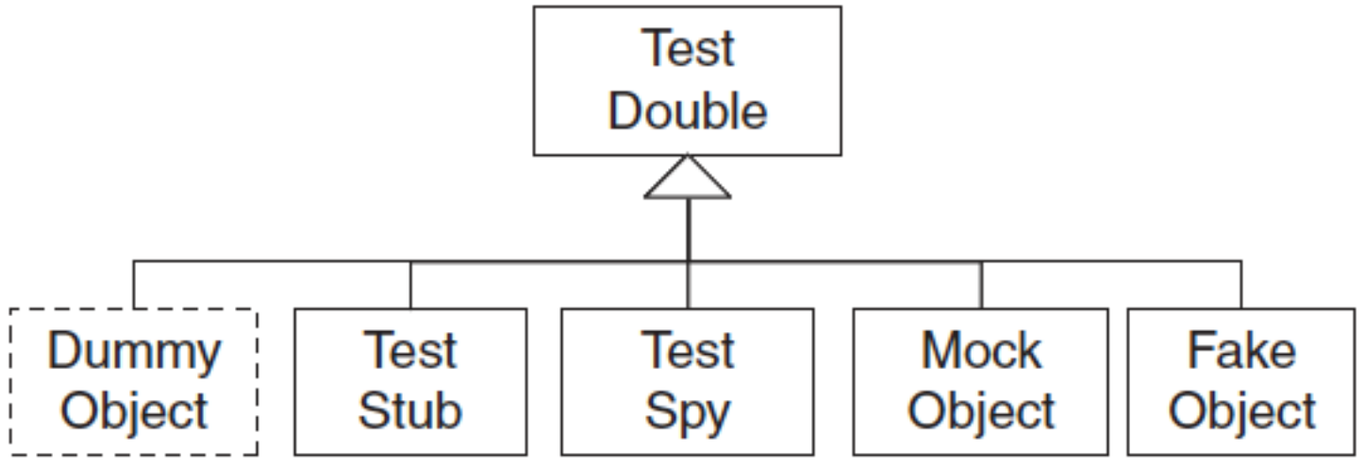
# Test Stub Motivation

```
public void testDisplayCurrentTime_AtMidnight()
{
    // Instantiate SUT (fixture setup)
    TimeDisplay sut = new TimeDisplay();

    // Exercise SUT
    String result = sut.getCurrentTimeAsHtmlFragment();

    // Verify outcome
    String expectedTimeString = "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals("Midnight", 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 Doubles – Topic List




- Understanding Test Double Terms (by example).
- Defining Test Doubles.
- Pattern Variations of Test Doubles.
- Test Stubs:
  - Hand-coded Test Stub.
  - Configurable Test Stub.
- Mock Objects:
  - Defining Mock Objects.
  - Mock Object Test Structure.
  - Maven.
  - Expectations.
  - ClassImposteriser.
- A note of caution on Test Doubles.

# Test Stub Hand-Code Example

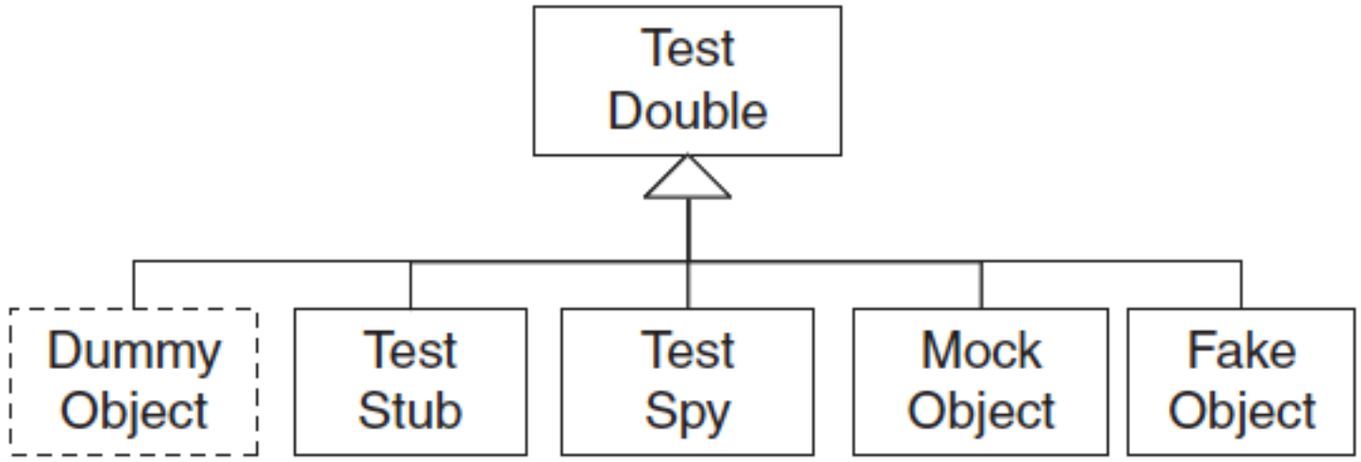
- TimeDisplay (SUT) depends on TimeProvider (DOC).
- The DOC is replaced with a stub - TimeProviderTestStub which is hard coded to return 00:00 time.
- This test stub is then used to inject indirect inputs into the SUT.

```
public void testDisplayCurrentTime_AtMidnight()
{
    // Fixture setup and Test Double configuration
    TimeProvider tpStub = new TimeProviderTestStub();  //hand coded stub
    tpStub.setHours(0);
    tpStub.setMinutes(0);

    // Instantiate SUT (fixture setup)
    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);
}
```

# Hand-coded Test Stub (`TimeProviderTestStub`)

```
//code omitted

private Calendar myTime = new GregorianCalendar();

public TimeProviderTestStub(int hours, int minutes) {
      setTime(hours, minutes);
}

public void setTime(int hours, int minutes) {
      setHours(hours);
      setMinutes(minutes);
}

public void setHours(int hours) {
      myTime.set(Calendar.HOUR_OF_DAY, hours);
}

public void setMinutes(int minutes) {
      myTime.set(Calendar.MINUTE, minutes); }

public Calendar getTime() {
      return myTime;
}

//code omitted
```

# Test Doubles – Topic List




- Understanding Test Double Terms (by example).
- Defining Test Doubles.
- Pattern Variations of Test Doubles.
- Test Stubs:
  - Hand-coded Test Stub.
  - Configurable Test Stub.
- Mock Objects:
  - Defining Mock Objects.
  - Mock Object Test Structure.
  - Maven.
  - Expectations.
  - ClassImposteriser.
- A note of caution on Test Doubles.

# Test Stub Using JMock Library

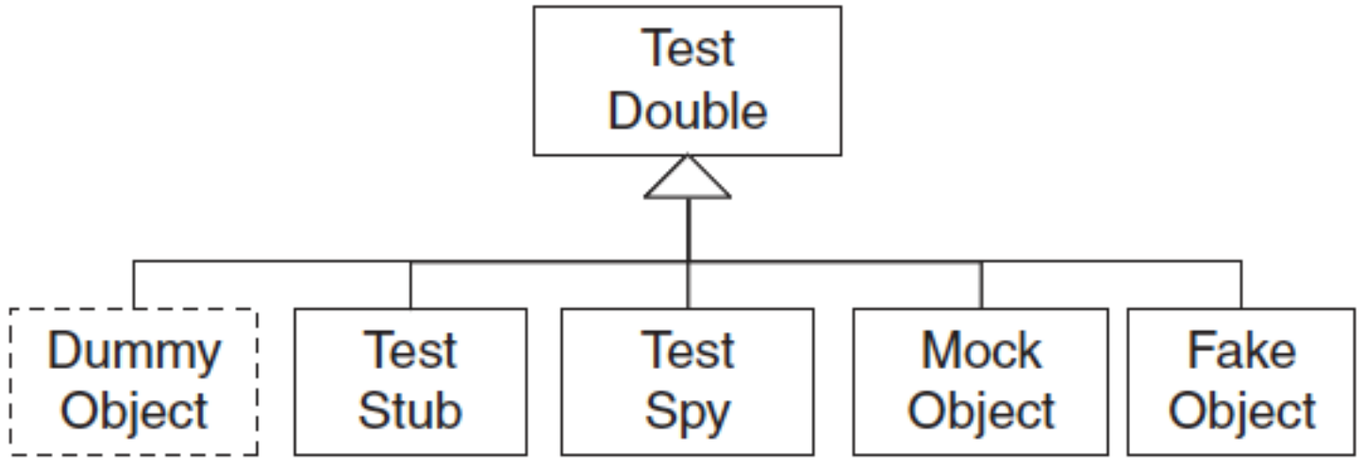
```
public void testDisplayCurrentTime_AtMidnight_JM()
{
    // Fixture setup
    TimeDisplay sut = new TimeDisplay();

    // Test Double configuration
    Mock tpStub = mock(TimeProvider.class);
    Calendar midnight = makeTime(0,0);              //makeTime is a test utility method
    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);
}
```

JMock generates *Mock Objects* dynamically. There is no Test Stub to hand code for this test…JMock framework implements the Test Stub for us.



http://www.jmock.org/getting-started.html

# Test Doubles – Topic List



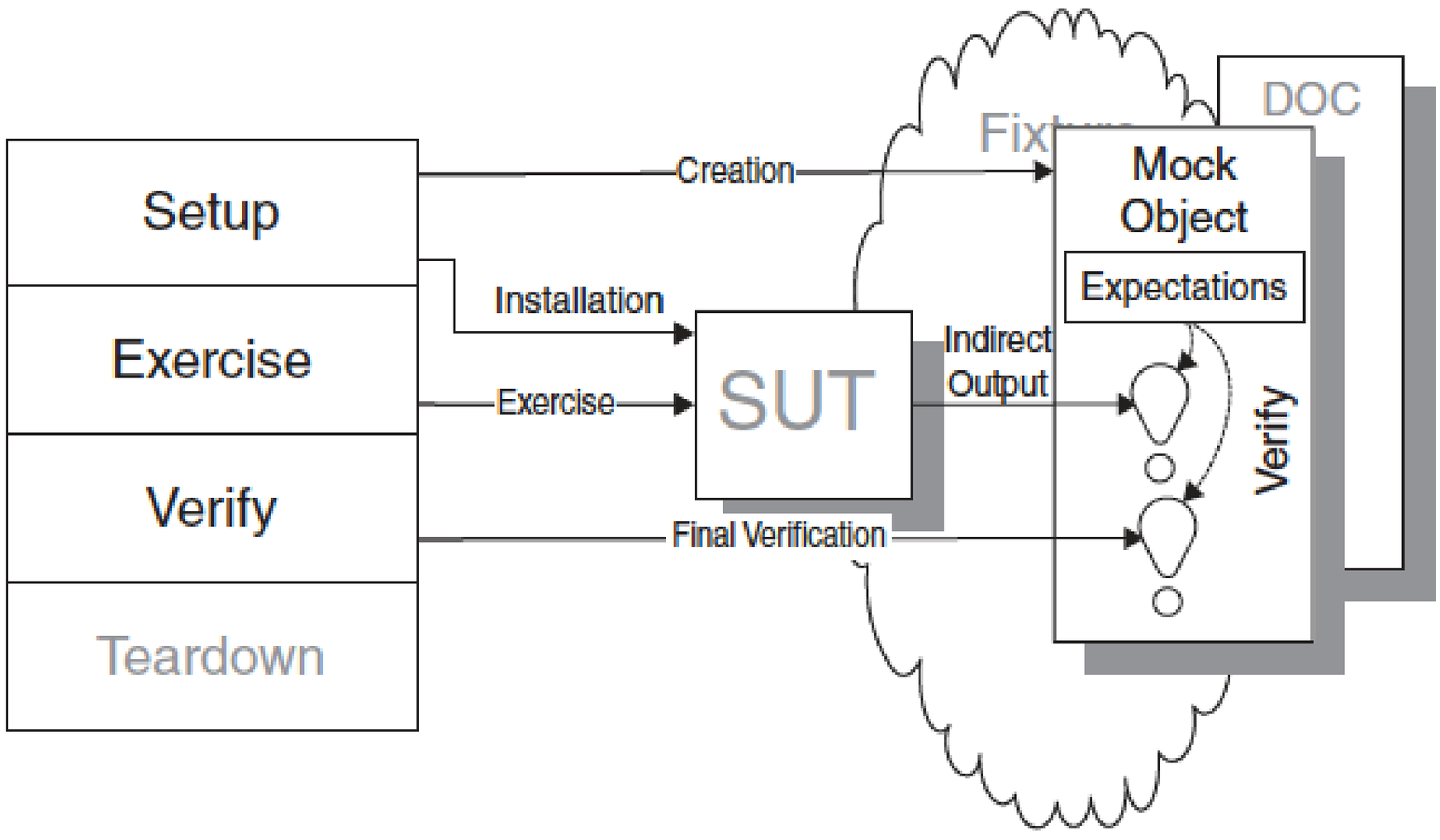

- Understanding Test Double Terms (by example).
- Defining Test Doubles.
- Pattern Variations of Test Doubles.
- Test Stubs:
  - Hand-coded Test Stub.
  - Configurable Test Stub.
- Mock Objects:
  - Defining Mock Objects.
  - Mock Object Test Structure.
  - Maven.
  - Expectations.
  - ClassImposteriser.
- A note of caution on Test Doubles.

# 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.



# Mock Object - 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.

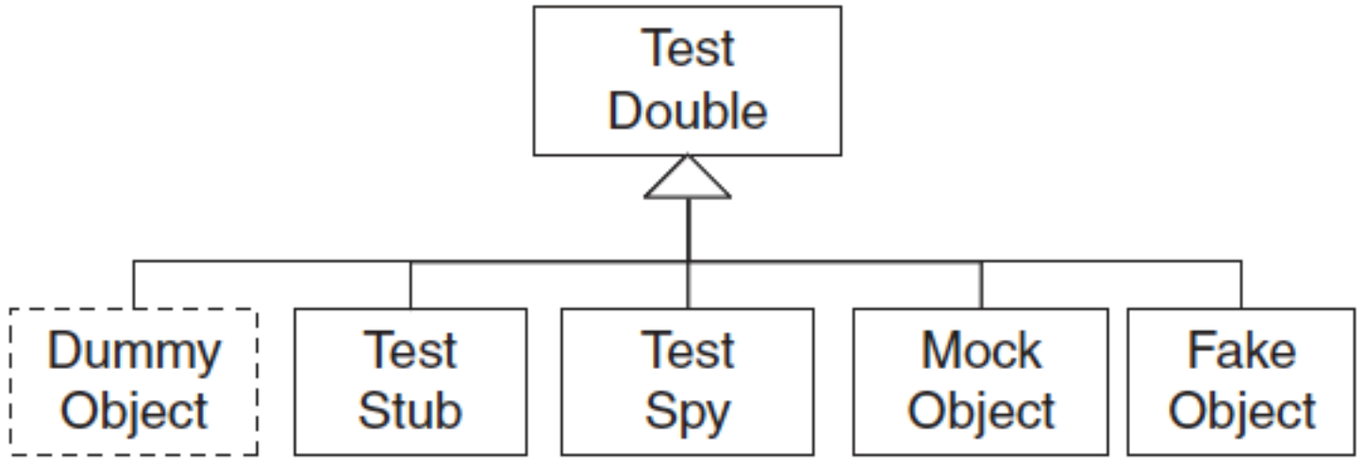
- When called during SUT execution, the Mock Object compares the actual arguments received with the expected arguments using equality assertions and fails the test if they don’t match.

# Mock Object - 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:
  - The fixture Setup phase of the test is broken down into a number of specific activities (next slide).
  - 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.

| Setup |
| --- |
| Exercise |
| Verify |
| Teardown |

# Test Doubles – Topic List




- Understanding Test Double Terms (by example).
- Defining Test Doubles.
- Pattern Variations of Test Doubles.
- Test Stubs:
  - Hand-coded Test Stub.
  - Configurable Test Stub.
- Mock Objects:
  - Defining Mock Objects.
  - Mock Object Test Structure.
  - Maven.
  - Expectations.
  - ClassImposteriser.
- A note of caution on Test Doubles.

# Mock Object - 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.

| Setup |
| --- |
| Exercise |
| Verify |
| Teardown |

# 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

- Having added the jMock jar files to your class path:
  - 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()
{
  //Fixture Setup
  Publisher publisher = new Publisher();
  final Subscriber subscriber = context.mock(Subscriber.class);
  publisher.add(subscriber);
  final String message = "message";


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

  //Exercise SUT
  publisher.publish(message);

  //Verify
  //context.assertIsSatisfied();
}
```

# 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

http://www.jmock.org/cookbook.html

# jMock 2 Cheat Sheet

http://www.jmock.org/cheat-sheet.html

## Test Fixture Class

```
import org.jmock.Mockery;
import org.jmock.Expectations;

public class AJMock2TestCase ... {
    Mockery context = new Mockery();

    ...
}
```

A `Mockery` represents the context of the object under test: the objects that it communicates with. Mockery is stored in an instance variable named *context*.

## Creating Mock Objects

```
Turtle turtle = context.mock(Turtle.class);
Turtle turtle2 = context.mock(Turtle.class, "turtle2");
```

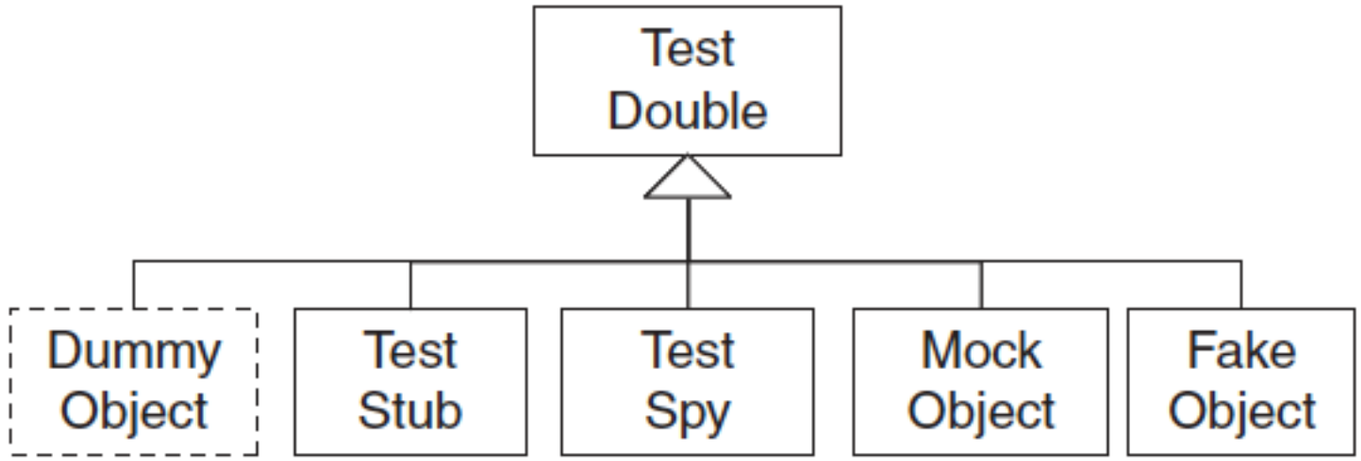
The examples above assume that the mock object is stored in an instance variable. If a mock ob
expectation blocks (see below).

## Tests with Expectations

```
... create mock objects ...

public void testSomeAction() {
    ... set up ...
```

Good article on Mocks and Stubs:
http://martinfowler.com/articles/mocksArentStubs.html

# Test Doubles – Topic List




- Understanding Test Double Terms (by example).
- Defining Test Doubles.
- Pattern Variations of Test Doubles.
- Test Stubs:
  - Hand-coded Test Stub.
  - Configurable Test Stub.
- Mock Objects:
  - Defining Mock Objects.
  - Mock Object Test Structure.
  - Maven.
  - Expectations.
  - ClassImposteriser.
- A note of caution on Test Doubles.

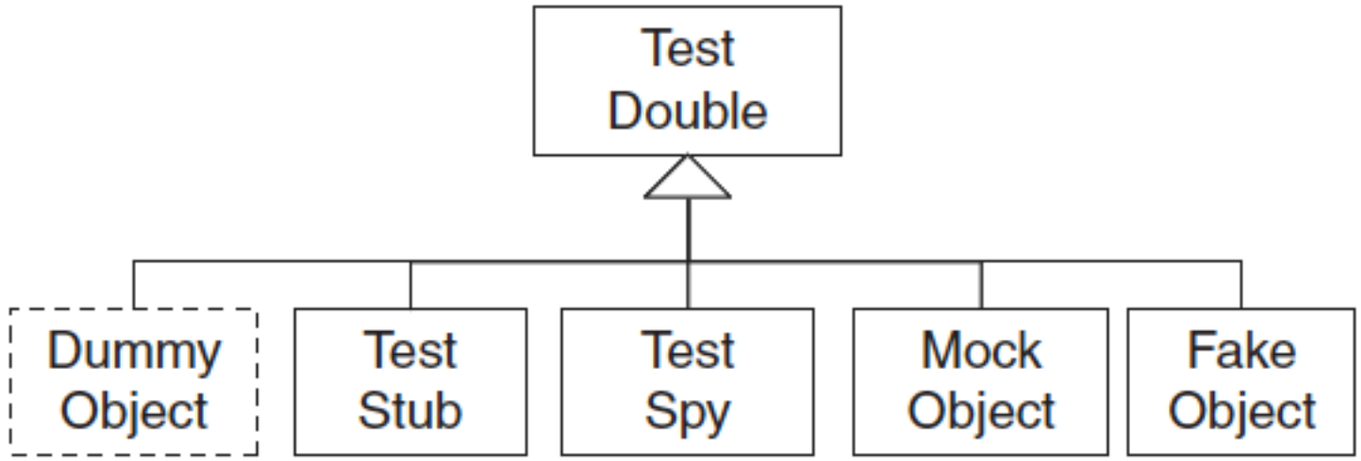
# Using jMock from Maven Builds

- The jMock jars are accessible via Maven by declaring the following dependency in your POM.

- All the required dependencies for jMock will be included automatically.

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

# Test Doubles – Topic List




- Understanding Test Double Terms (by example).
- Defining Test Doubles.
- Pattern Variations of Test Doubles.
- Test Stubs:
  - Hand-coded Test Stub.
  - Configurable Test Stub.
- Mock Objects:
  - Defining Mock Objects.
  - Mock Object Test Structure.
  - Maven.
  - Expectations.
  - ClassImposteriser.
- A note of caution on Test Doubles.

# A note on Expectations

- Expectations are defined within a "Double-Brace Block“ i.e. {{ }}

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

- It defines the expectations in the context of the test's Mockery.
- An expectations block can contain any number of expectations.
- Each expectation has the following structure:

```
invocation-count (mock-object).method(argument-constraints);
    inSequence(sequence-name);
    when(state-machine.is(state-name));
    will(action);
    then(state-machine.is(new-state-name));
```

- Except for the invocation count and the mock object, all clauses are optional.

# A note on Expectations - examples

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

```
oneOf (turtle).turn(45);           // The turtle will be told to turn 45 degrees once only

allowing (turtle).flashLEDs();     // The turtle can be told to flash its LEDs any number of
                                   //types or not at all

ignoring(turtle2);                 // Turtle 2 can be told to do anything. This test ignores
                                   //it.

allowing (turtle).queryPen();      // The turtle can be asked about its pen any number of
  will(returnValue(PEN_DOWN));     // times and will always return PEN_DOWN

atLeast(1).of (turtle).stop();     // The turtle will be told to stop at least once.
```

# Invocation Count

| | |
|---|---|
| oneOf | The invocation is expected once and once only. |
| exactly(*n*).of | The invocation is expected exactly *n* times.<br>Note: one is a convenient shorthand<br>for exactly(1). |
| atLeast(*n*).of | The invocation is expected at least *n*. |
| 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. |

# Returning Values

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

```
oneOf (calculator).add(1, 1); will(returnValue(2));
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:

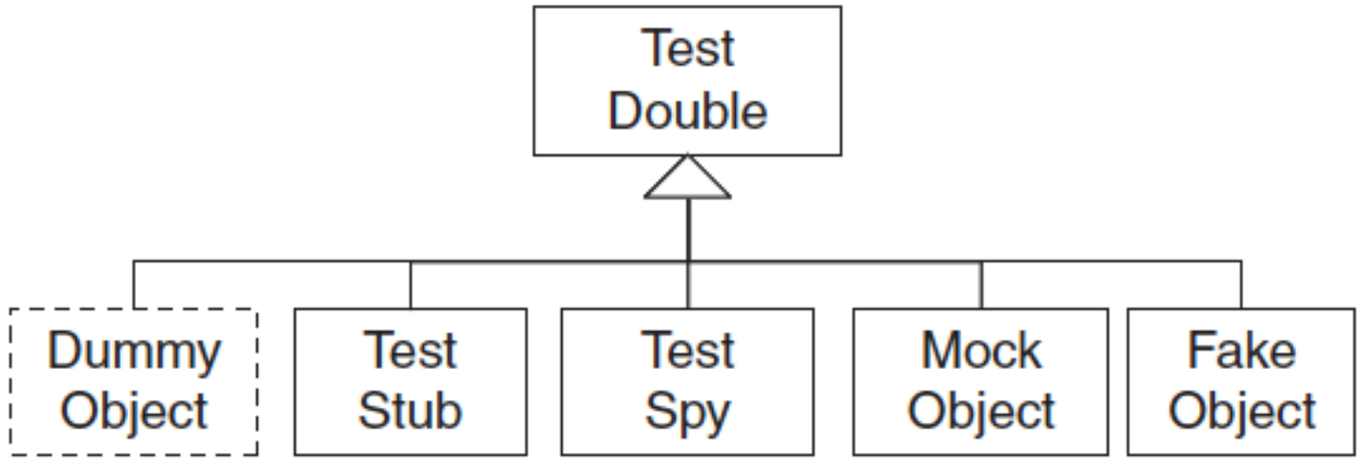
```
context.checking(new Expectations()
{{
   oneOf (department).employees(); will(returnIterator(alice, bob));
}});
```

# Exceptions

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

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

# Test Doubles – Topic List



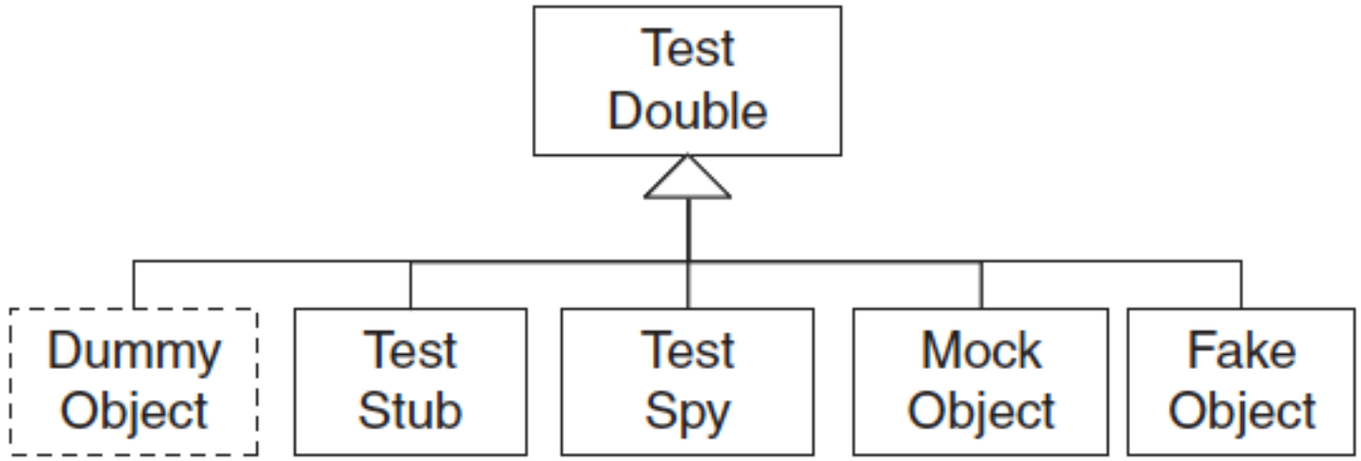

- Understanding Test Double Terms (by example).
- Defining Test Doubles.
- Pattern Variations of Test Doubles.
- Test Stubs:
  - Hand-coded Test Stub.
  - Configurable Test Stub.
- Mock Objects:
  - Defining Mock Objects.
  - Mock Object Test Structure.
  - Maven.
  - Expectations.
  - ClassImposteriser.
- A note of caution on Test Doubles.

# Mocking Classes with jMock and the ClassImposteriser

- The default configuration of the jMock framework can only mock interfaces, not classes.
- However, the ClassImposteriser extension class uses the CGLIB 2.1 and Objenesis 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
  }

}
```

# Test Doubles – Topic List




- Understanding Test Double Terms (by example).
- Defining Test Doubles.
- Pattern Variations of Test Doubles.
- Test Stubs:
  - Hand-coded Test Stub.
  - Configurable Test Stub.
- Mock Objects:
  - Defining Mock Objects.
  - Mock Object Test Structure.
  - Maven.
  - Expectations.
  - ClassImposteriser.
- A note of caution on Test Doubles.

# A word of caution on Test Doubles

- Exercise caution when using *Test Doubles:*

  → you are testing the SUT in a different configuration from that which will be used in production.

  →Have at least one test that verifies it works without a *Test Double*.

  →be careful you don't replace the parts of the SUT that you are trying to verify as this can result in tests that test the wrong software!

  →excessive use of *Test Doubles* can result in *Fragile Tests* as a result of *Overspecified Software* (i.e. a test says too much about how the software should be structured or behave).