Unit Testing

Venkatesh-Prasad Ranganath
Kansas State University

Slides with * in title capture whiteboard content

Unit Testing Terms

- *Test*
- *UUT / SUT*
- *Test case*
  - *Structure*
    - *Test Data and Test Logic*
    - *Assertions*
- *Test Suite*
- *Test Fixture*
  - *Setup*
  - *Teardown*
- *Test Runner*
- *Testing Framework*

Unit Testing Terms

- *Test(ing)* - The act of exercising software with test cases
- *UUT / SUT* - Unit / System Under Test
- *Test Case* - An identifiable unit of testing; often, bundled as a method/function
  - *Structure* - Setup, Execution, Verify, Teardown (Arrange, Act, Assert)
- *Test Data and Test Logic* - Integral components of a test case
- *Assertion* - Preferred mechanism to state/communicate test outcome

```
if <actual outcome is not the same as expected outcome> then
    raise AssertionError()  # test failed
else
    pass
```

Unit Testing Terms

- *Test Suite* - A collection of test cases
- *Test Fixture*
  - *Setup* - Preparation to be performed before tests
  - *Teardown* - Cleanup to be performed after tests
- *Test Runner* - A component that orchestrates the execution of test cases and collects the test verdicts.
- *Testing Framework* - A framework that supports test automation; typically, includes test runner, reporting capability, coverage calculation, and other features.

XUnit Testing Frameworks in Different Languages

Python

- TestCar — test class name
- test_drive — test method name

- unittest, Nose, PyTest
- Mostly based on convention and consistent practice

Java

- CarTest
- testDrive
- @Test — Test method annotation

- JUnit, TestNG, Spock
- Mostly based on convention and consistent practice

Common wisdom in unit testing

- Many test cases for one method/function
- Many test classes for one class

Coding Session*

checkId(x) Specification

- x is a string
  - If not, raise TypeError
- x represents a numerical value
  - If not, raise ValueError
- True if x has 9 characters
- False otherwise

Test Runner

How does a test runner work?

Test Runner

How does a test runner work?

- find/discover test cases
  - based on rules or conventions or annotations
    - E.g., functions whose name starts with “test”
- execute each test case
- collect and compile the test verdicts

Testing Framework

How do most XUnit testing framework determine test verdict?

Testing Framework

How do most XUnit testing framework determine test verdict?

- *Failure* if test throws exception X
- *Pass* if test completes (without any exception / failures)
- *Error* otherwise

- Most XUnit framework rely on assert statements and use the corresponding exception (e.g., AssertionError) as exception X

Assertions

- What's the deal with assertion?
- What are the pros and cons of using assertions?
- With regards to inputs/outputs, when to use assertion and when to use error handling?

Assertions

What's the deal with assertion?

- Assertion is a mechanism to check if (should-be-true) properties are true; typically, guarantees
  - E.g., input array to a binary search function should be sorted
  - E.g., output array returned by a sorting function should be of the same length as the input array

What are the pros and cons of using assertions?

- + Easily and succinctly add checks to programs
- - Imposes performance penalties

Assertions

With regards to inputs, when to use assertion and when to use error checking? (Guidance)

- Use error checking to handle both valid and invalid inputs and generate corresponding outcome; when both valid and invalid inputs are expected.
  - E.g., sq_root will return the square root of the input integer if it is positive; otherwise, raise ValueError exception.
  - Input provider does not promise to ensure the validity of input.
- Use assertions to check inputs are valid; when only valid inputs are expected.
  - E.g., sq_root will return square root of input integer and it should be invoked only with positive integers.
  - Input provider promises to provide the validity of input.

Test Case Structure

Setup, *Execution, Verify*, Teardown (Arrange, Act, Assert)

- What do we do in Execution?
- What do we do in Verify?
- What is the common code pattern to test for outcome and absence/presence of exceptional behavior?
- Setup and Teardown are considered part of Test Fixture

Test Case Structure

- What do we do in Execution?
  - Set up data/objects necessary to obtain actual outcome and observe the outcome
- What do we do in Verify?
  - Compare the observed outcome to expected outcome

Test Case Structure

What is the common code pattern to test for output?

```
<action_1>
<action_2>
…
output = <action_n>
assert output == <expected_output>
```

We use an appropriate operator in place of == to compare observed output and expected output

Test Case Structure

What is the common code pattern to test for absence of exceptional behavior/outcome?

```
try:
    <action_1>
    <action_2>
    …
    <action_n>
except:
    assert False
```

Test Case Structure

What is the common code pattern to test for presence of exceptional behavior/outcome?

```
try:
    <action_1>
    <action_2>
    …
    assert False
except:
    pass
```

Test Case Structure

What is the common code pattern to test for presence of specific exceptional behavior/outcome, i.e., raises exception X?

```
try:
    <action_1>
    <action_2>
    …
    assert False
except X:
    pass
except:
    assert False
```

Test Fixture

What should we do in Setup?

- ??

What should we do in Teardown?

- ??

Test Fixture

What should we do in Setup?

- Put the UUT in the state required for testing
- Perform actions common to a set of test cases

What should we do in Teardown?

- Perform clean up actions
  - Release external resources
  - Delete interim artifacts
  - Restore system back to a good state

Parameterized Unit Tests

```
void TestAdd() {
   ArrayList a = new ArrayList(0);
   object o = new object();
   a.Add(o);
   Assert.IsTrue(a[0] == o);
}
```

- Parameterized Unit Tests by Nikolai Tillmann and Wolfram Schultz [FSE'05] (Section 2)

Parameterized Unit Tests

```
void TestAdd() {
   ArrayList a = new ArrayList(0);
   object o = new object();
   a.Add(o);
   Assert.IsTrue(a[0] == o);
}
void TestAdd(ArrayList a, object o) {
   if (a != null) {
      int i = a.Count;
      a.Add(o);
      Assert.IsTrue(a[i] == o);
   }
}
```

- Parameterized Unit Tests by Nikolai Tillmann and Wolfram Schultz [FSE'05] (Section 2)

Parameterized Unit Tests

```
void TestAdd() {
   ArrayList a = new ArrayList(0);
   object o = new object();
   a.Add(o);
   Assert.IsTrue(a[0] == o);
}
void TestAdd(ArrayList a, object o) {
   Assume.IsTrue(a != null);
   int i = a.Count;
   a.Add(o);
   Assert.IsTrue(a[i] == o);
}
```

- Parameterized Unit Tests by Nikolai Tillmann and Wolfram Schultz [FSE'05] (Section 2)

Parameterized Unit Tests

- PUTs are test methods/cases generalized by allowing parameters
  - Most often, parameters correspond to test data (as in example-based testing)
- PUTs are not test cases
- PUTs are templates of test cases
  - Test cases are instantiations of PUTs with specific arguments (inputs)

Test Structure

Example-based Test Case

1. Setup
2. Execute
3. Verify
4. Teardown

Parameterized Unit Tests

1. Check assumptions about parameters/inputs
2. Setup
3. Execute
4. Verify
5. Teardown

Test Doubles (Harness)

What is a test double?

- xUnit Test Patterns by Gerard Meszaros (Chapter 23)

Test Doubles (Harness)

What is a test double?

- SUT may depend on components (DOCs) that are not under test
- Test doubles are replacements for DOCs

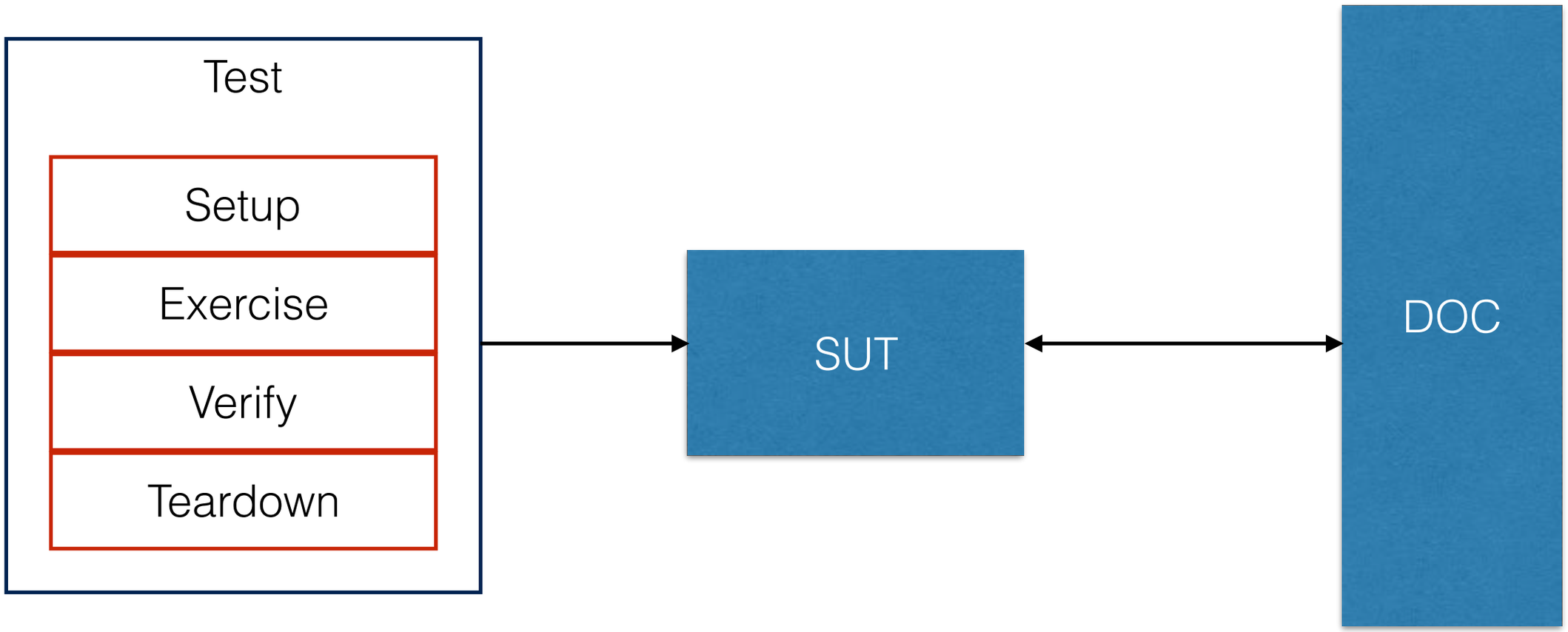
- xUnit Test Patterns by Gerard Meszaros (Chapter 23)

Why use Test Doubles?

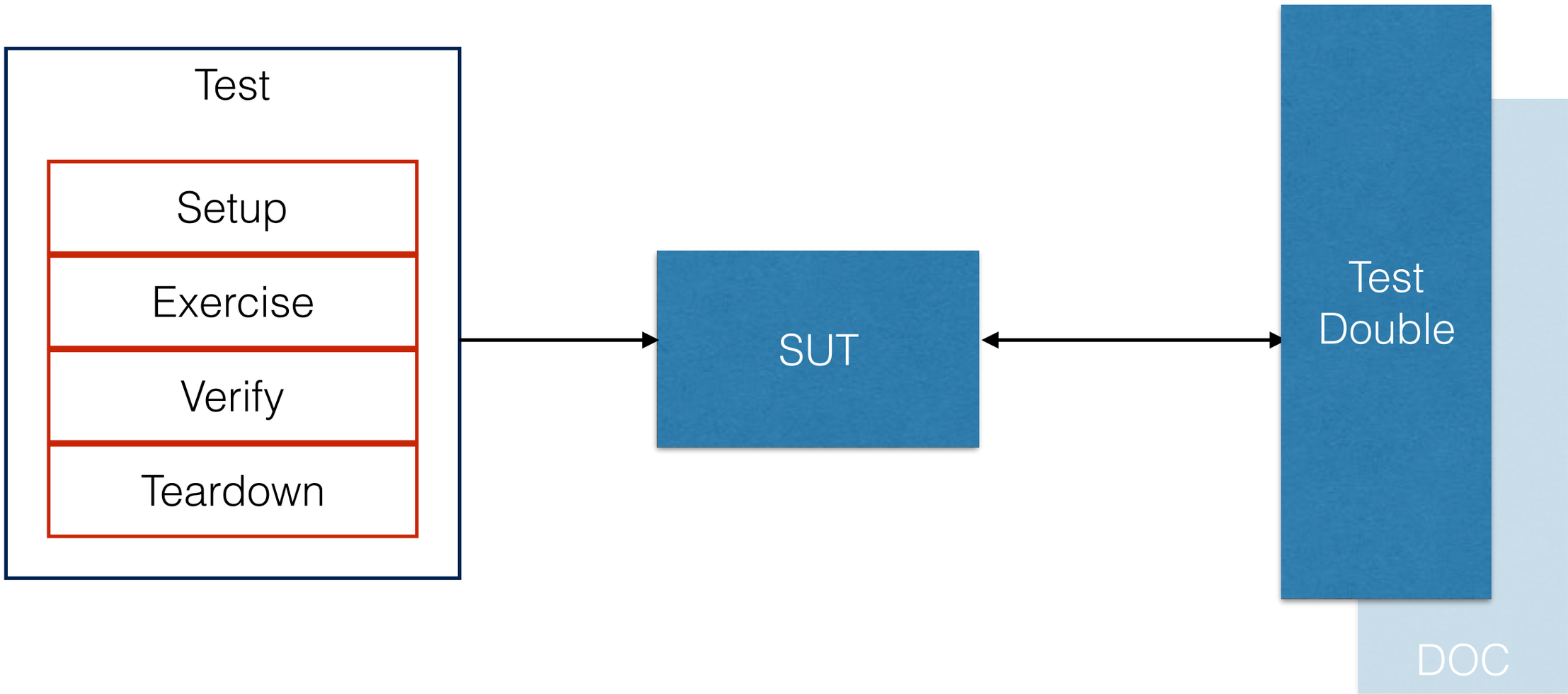
Why use Test Doubles?

- DOC is overly large / long setup time
- DOC is slow
- Do not want to mess existing DOC
- DOC is unavailable or not ready
- Test dependency on DOC interface (not implementation)

Usual Setup



Test Doubles (Harness)



- xUnit Test Patterns by Gerard Meszaros (Chapter 23)

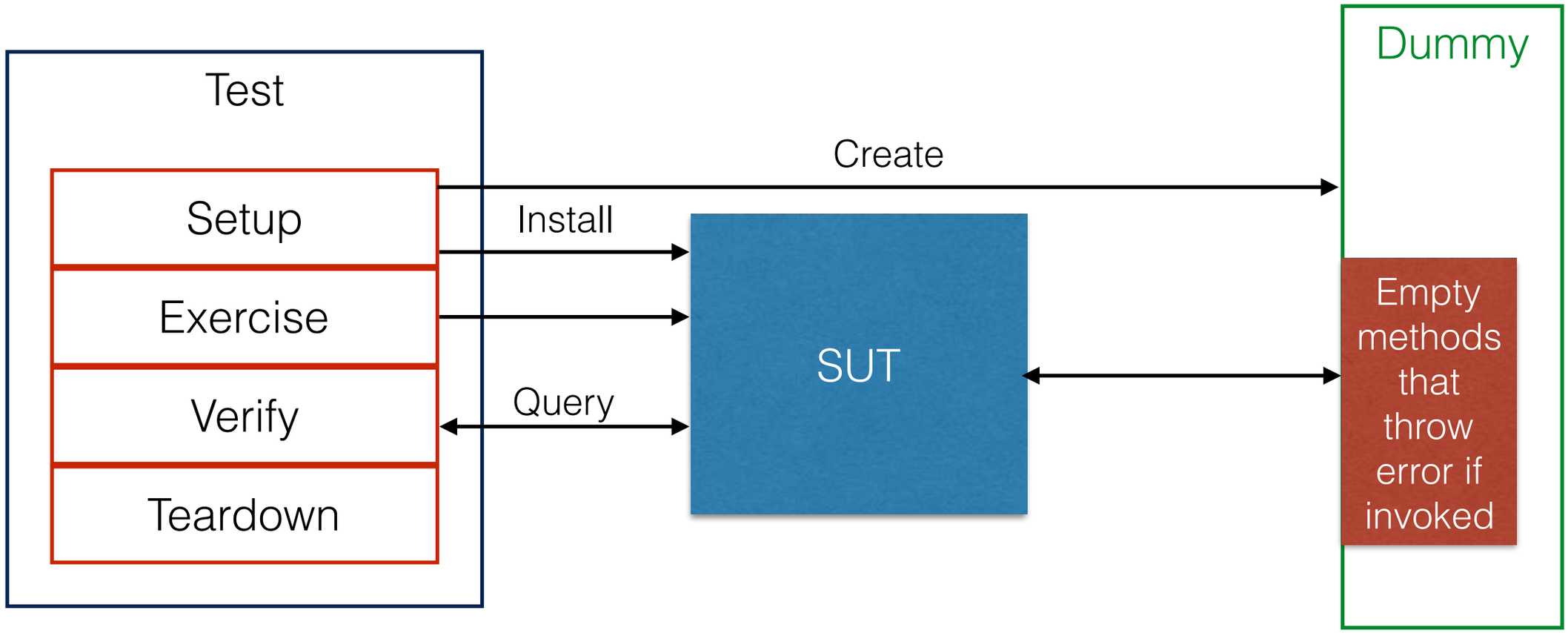
Test Doubles (Harness)

What are the different sorts of test doubles?

- Dummy
- Fake
- (Test) Stub
- Test Spy
- Mock Object

- xUnit Test Patterns by Gerard Meszaros (Chapter 23)

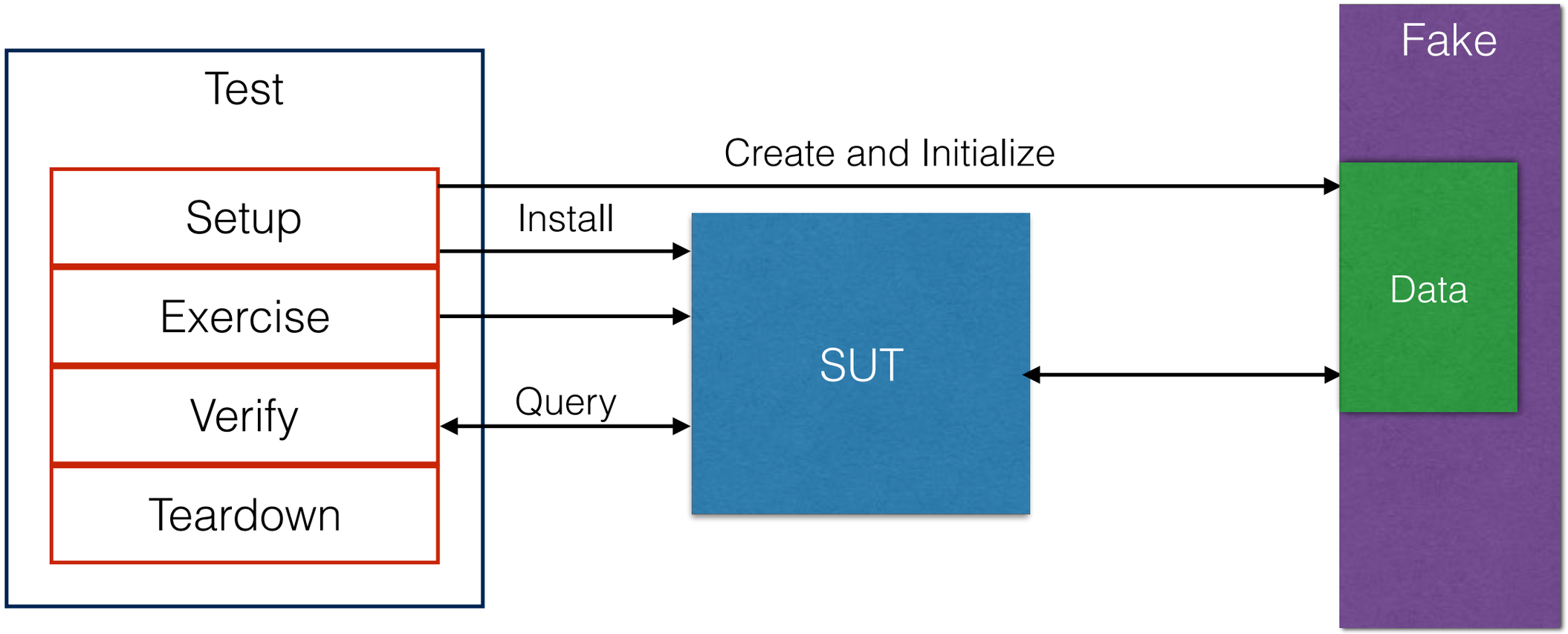
Dummy



How do we specify the values to be used in tests when their only usage is as irrelevant arguments to SUT method calls?
We pass an object that has no implementation as an argument of a method called on the SUT.

- xUnit Test Patterns by Gerard Meszaros (Chapter 27)

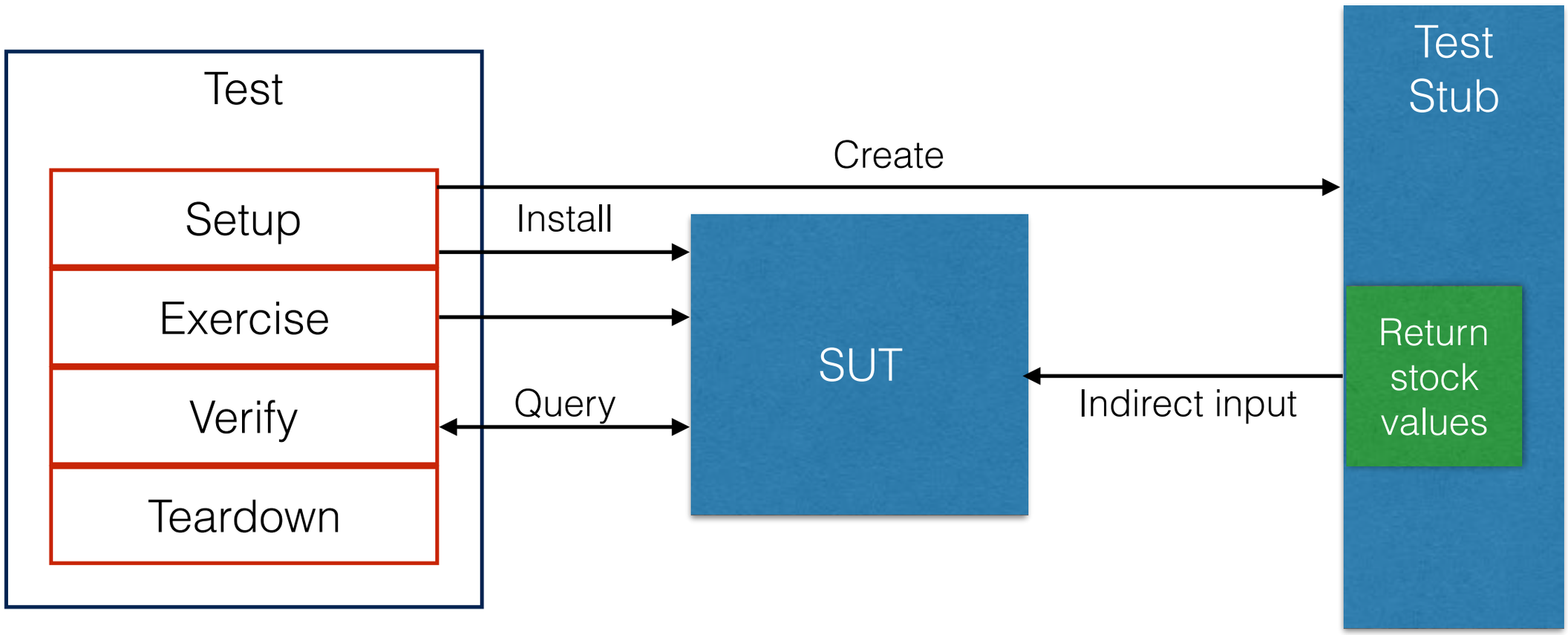
Fake



How can we verify logic independently when depended-on objects cannot be used? How can we avoid slow tests?
We replace a component that the SUT depends on with a much lighter-weight implementation.

- xUnit Test Patterns by Gerard Meszaros (Chapter 23)

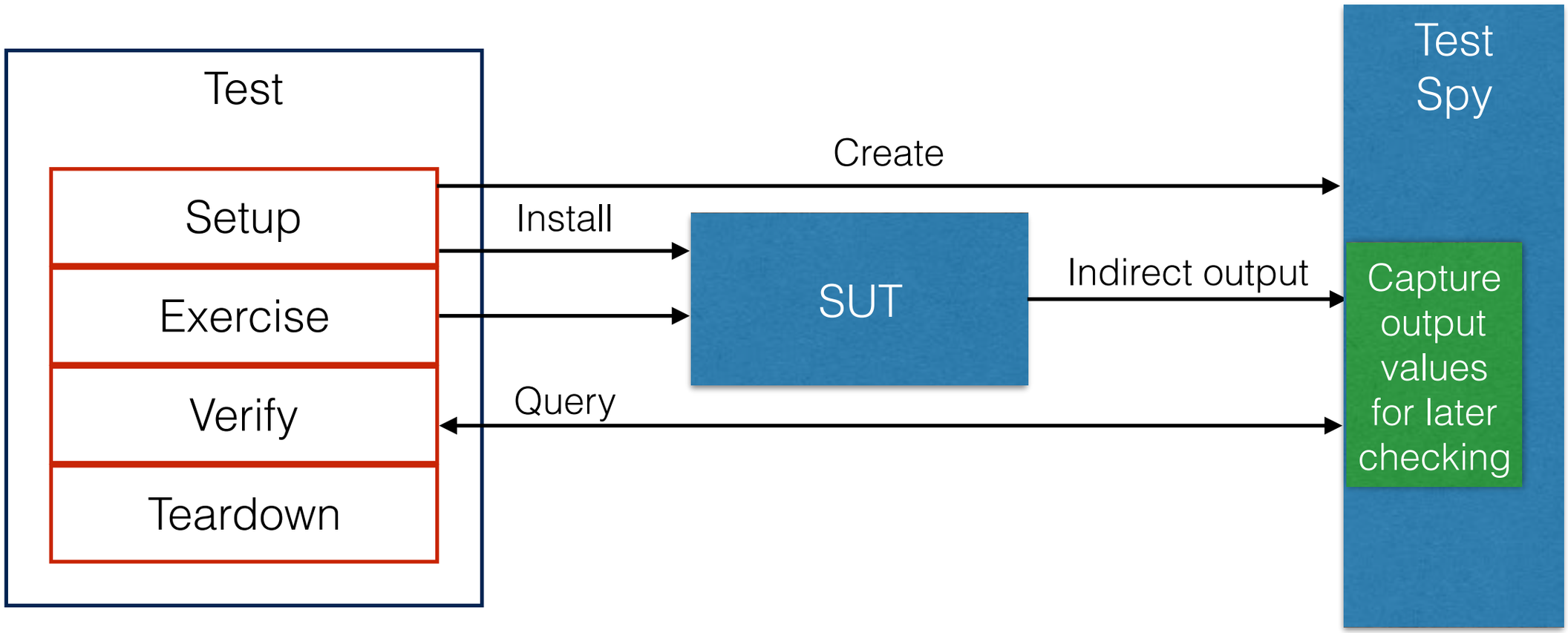
Test Stub



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.

- xUnit Test Patterns by Gerard Meszaros (Chapter 23)

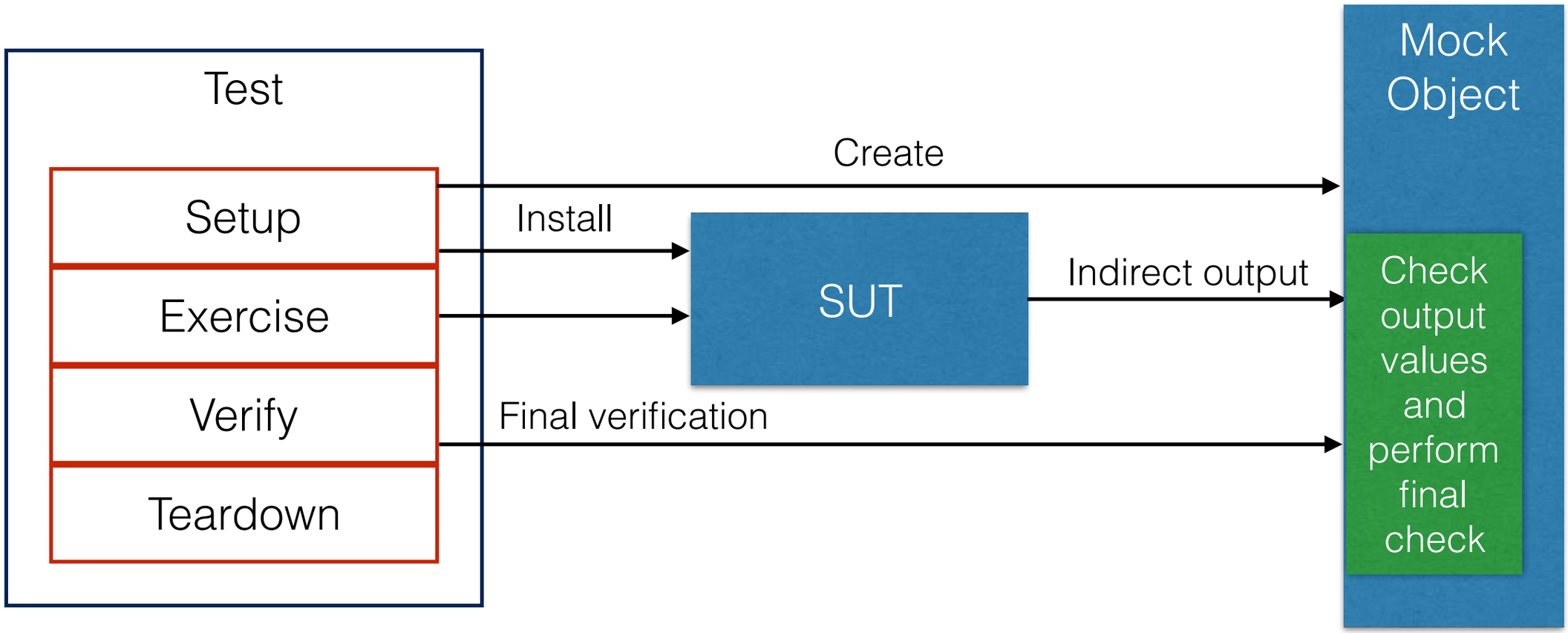
Test Spy



How do we implement *behavior verification*? How can we verify logic independently when it has indirect outputs to other software components?
We use a test double to capture the indirect output calls made to another component by the SUT for later verification by the test.

- xUnit Test Patterns by Gerard Meszaros (Chapter 23)

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.

- xUnit Test Patterns by Gerard Meszaros (Chapter 23)