Seminar 8 - Mockito

PV260 Software Quality

April 9, 2015

Mocking in Unit Testing

- Unit testing is simple for classes with no dependencies
- How do we test an object which depends on many other things (many of which might not even be implemented yet) ?
- We create stand-in objects which share interface with the required dependency
- Inside, instead of some complex behavior, these are hard-wired to work in the one particular test case
- We can create these substitutes either by hand or use a mocking framework

Mockito

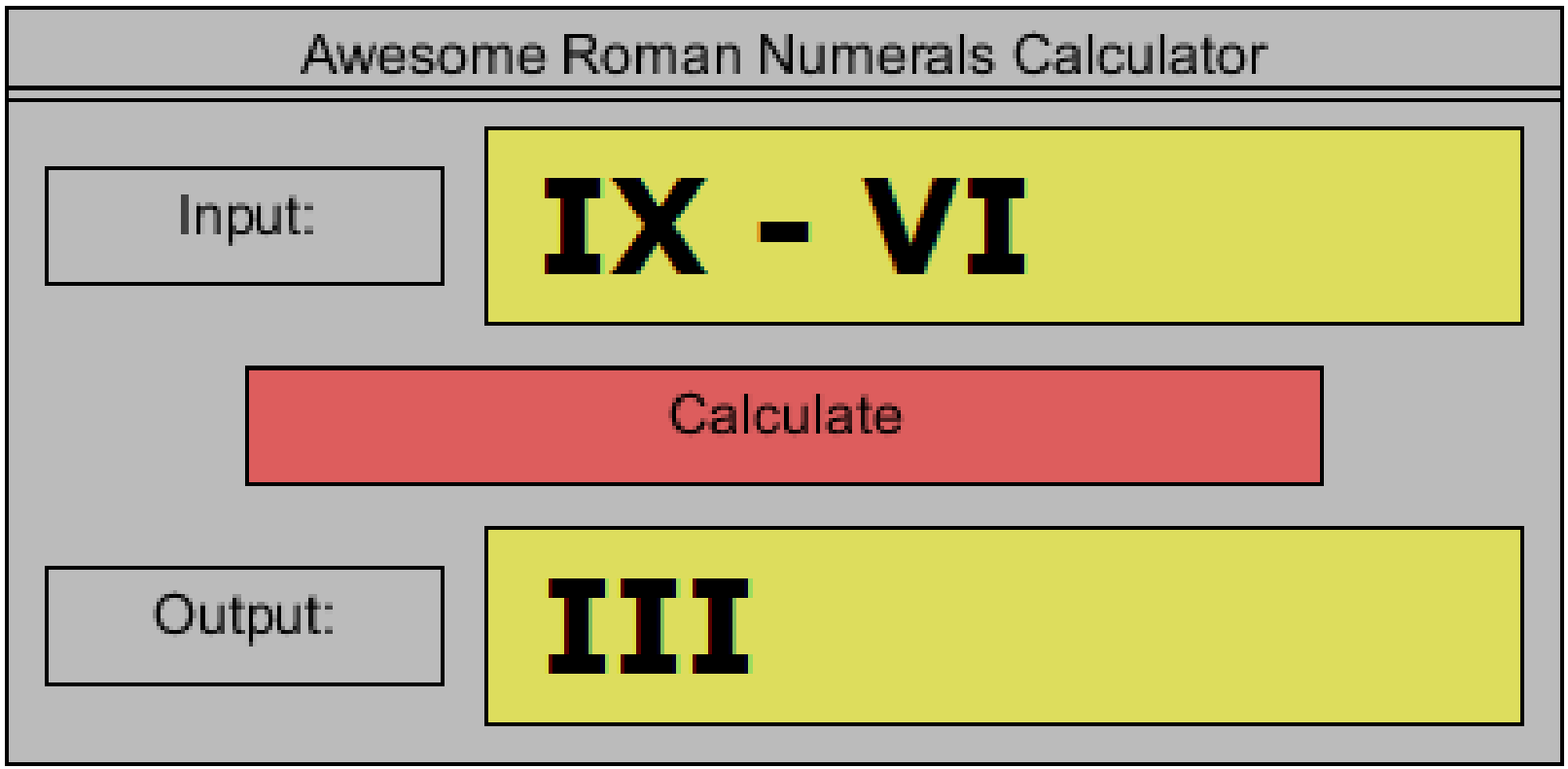
http://mockito.org/

> *We decided during the main conference that we should use JUnit 4 and Mockito because we think they are the future of TDD and mocking in Java.*
> *(Dan North - author of BDD)*

- Interaction verification
- Input stubbing (data, exceptions. . .)
- Test Spy wrappers
- Mock both classes and interfaces
- Lightweight API

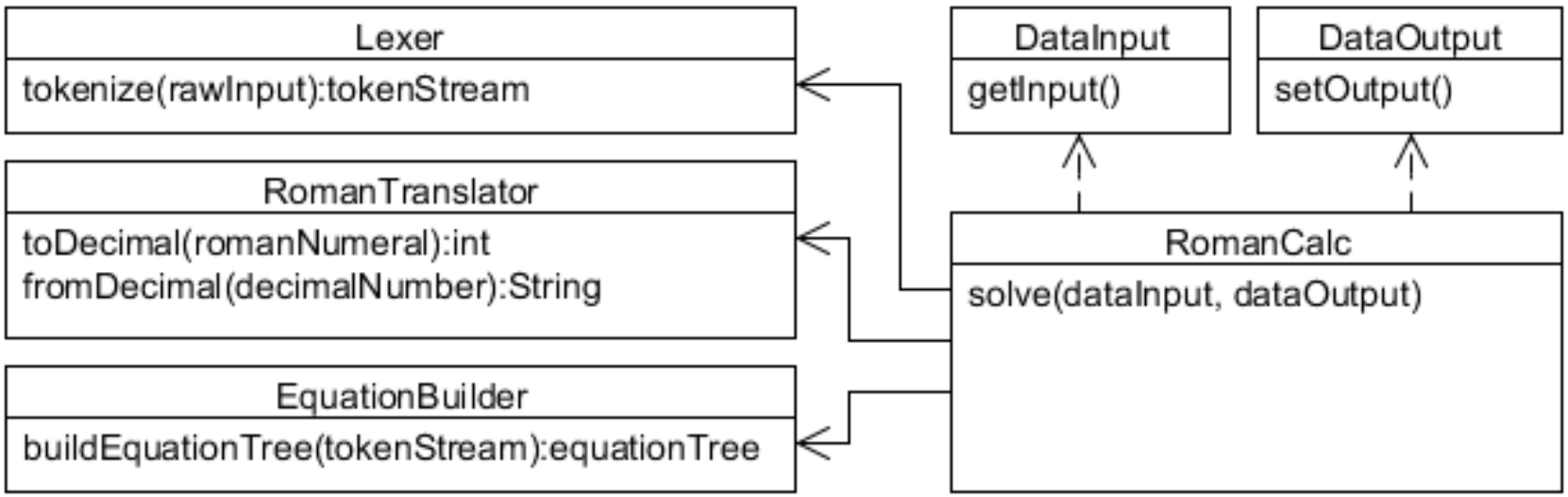
Working Example

- Model for an app doing basic math on Roman numerals
- We only care about the inner logic, the UI doesn't concern us



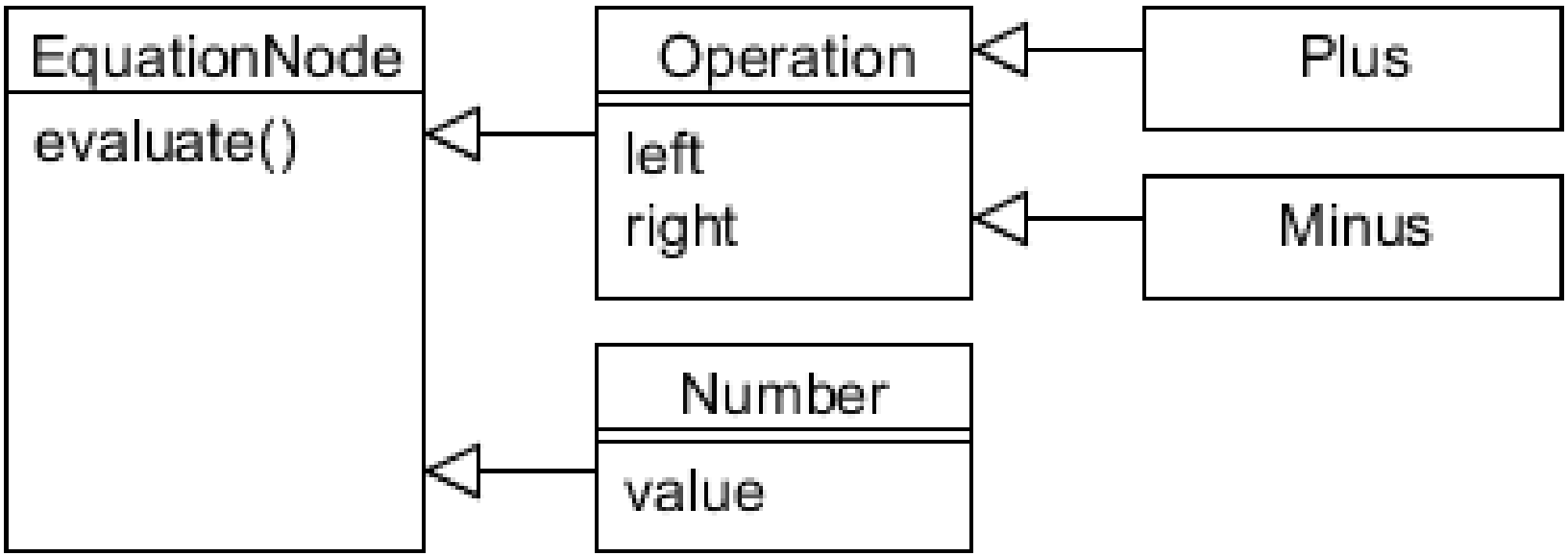
Working Example - Structure

- We already have the design done, all interfaces are prepared
- DataInput and DataOutput represent the textboxes
- Clicking the Calculate button calls the solve method



Working Example - Structure

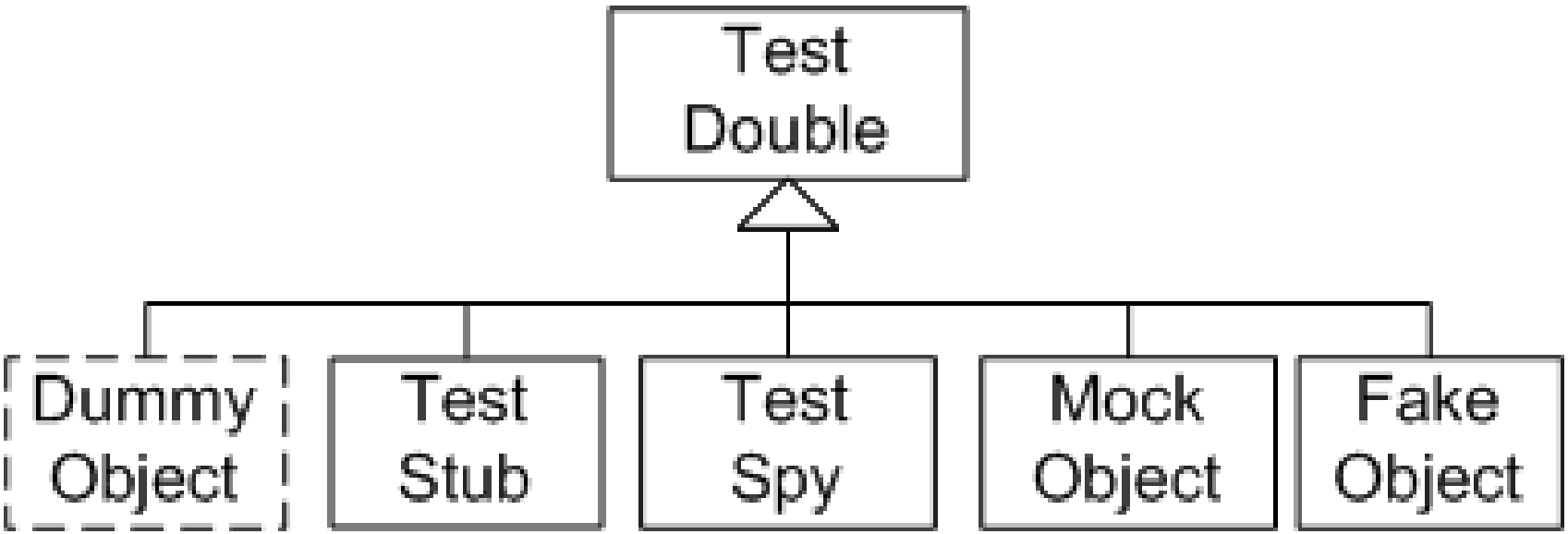
- Lexer tokenizes the raw input
- Number tokens are translated by the RomanTranslator and sent to EquationBuilder
- Tree representation of the equation is assembled
- The decimal result is translated to Roman numerals
- Formated result is sent back to output



Test Doubles Hierarchy

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

- There are many types of stand-in objects used in testing
- Each plays a different role, the simplest type possible should be used (That is dont use a Mock if all you need is a Dummy)



Dummy Object

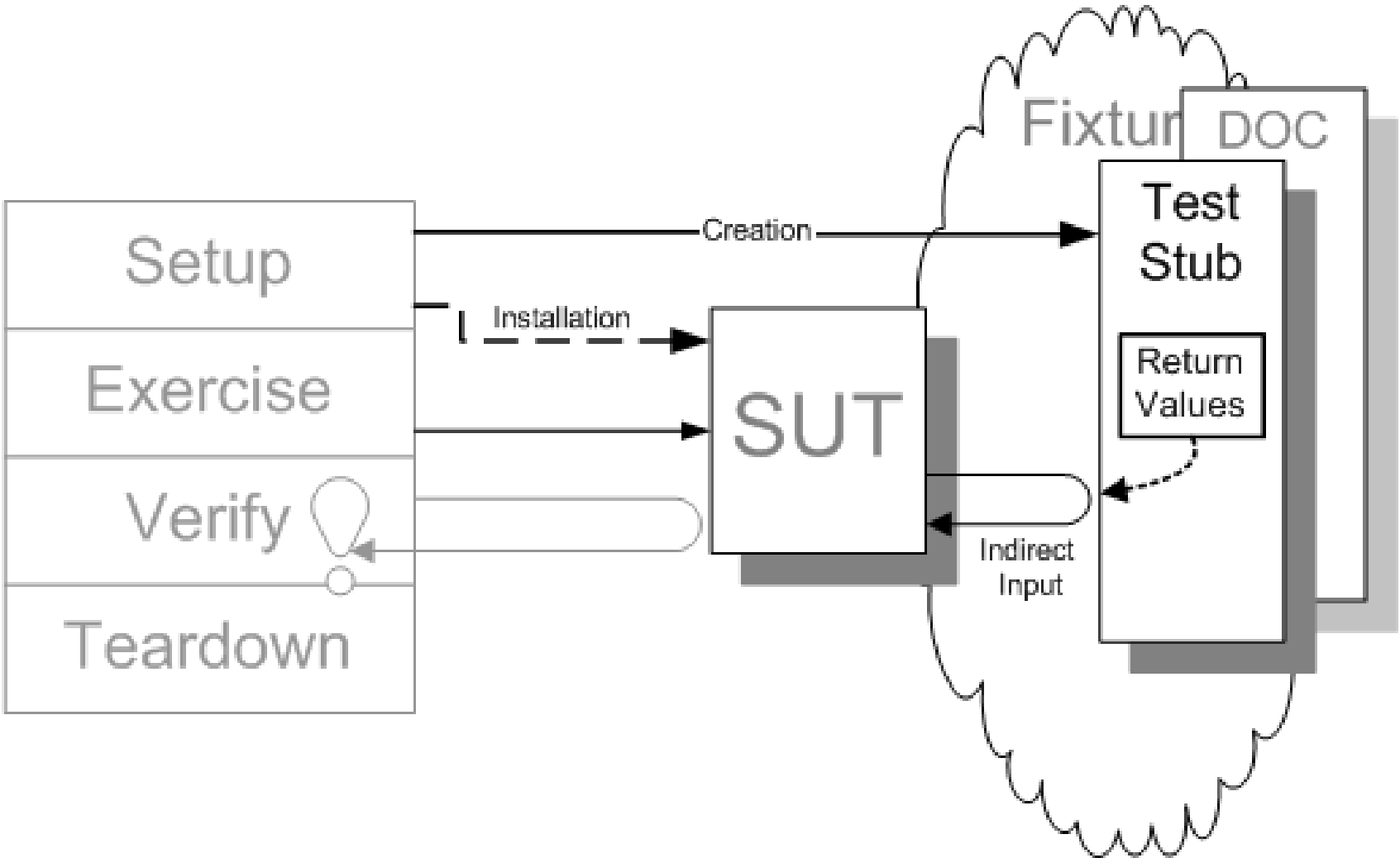
RomanCalculatorTest#testExceptionFromInput

- We need to provide real object (that is not *null*), but at the same time we know it will never be used during the test
- Even better, we pass *null* to the test which helps readability as we are clearly signalling that the value is not used
- This is of course not possible with null-checks in constructors, so we have to use dummies instead.

Test Stub

RomanTranslatingTokenStream#testConvertsToDecimalTokens

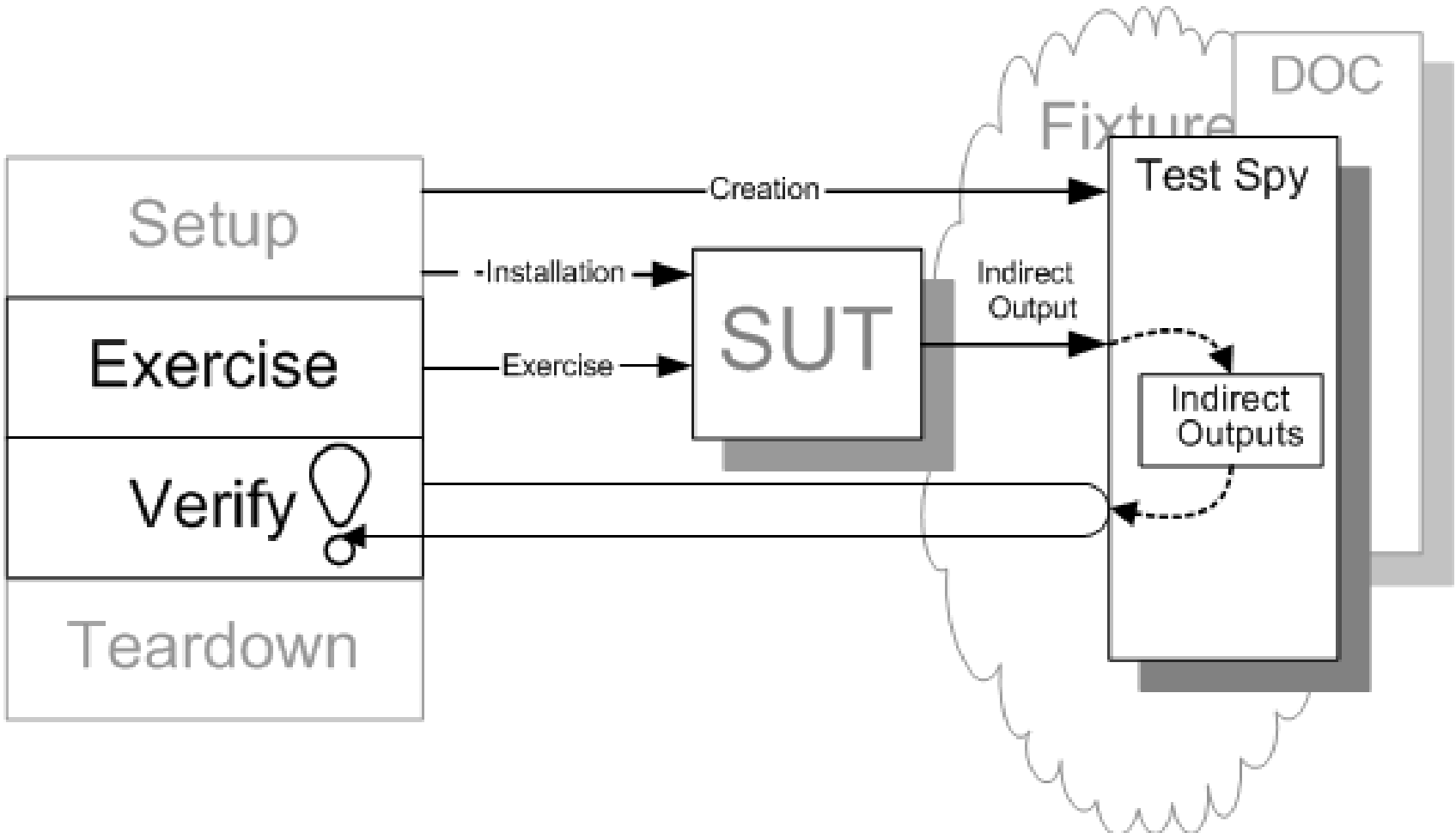
- We want one of SUT's dependencies to provide specific input to the SUT when queried



Test Spy

RomanTranslatingTokenStream#testRecognizesRomanNumeral

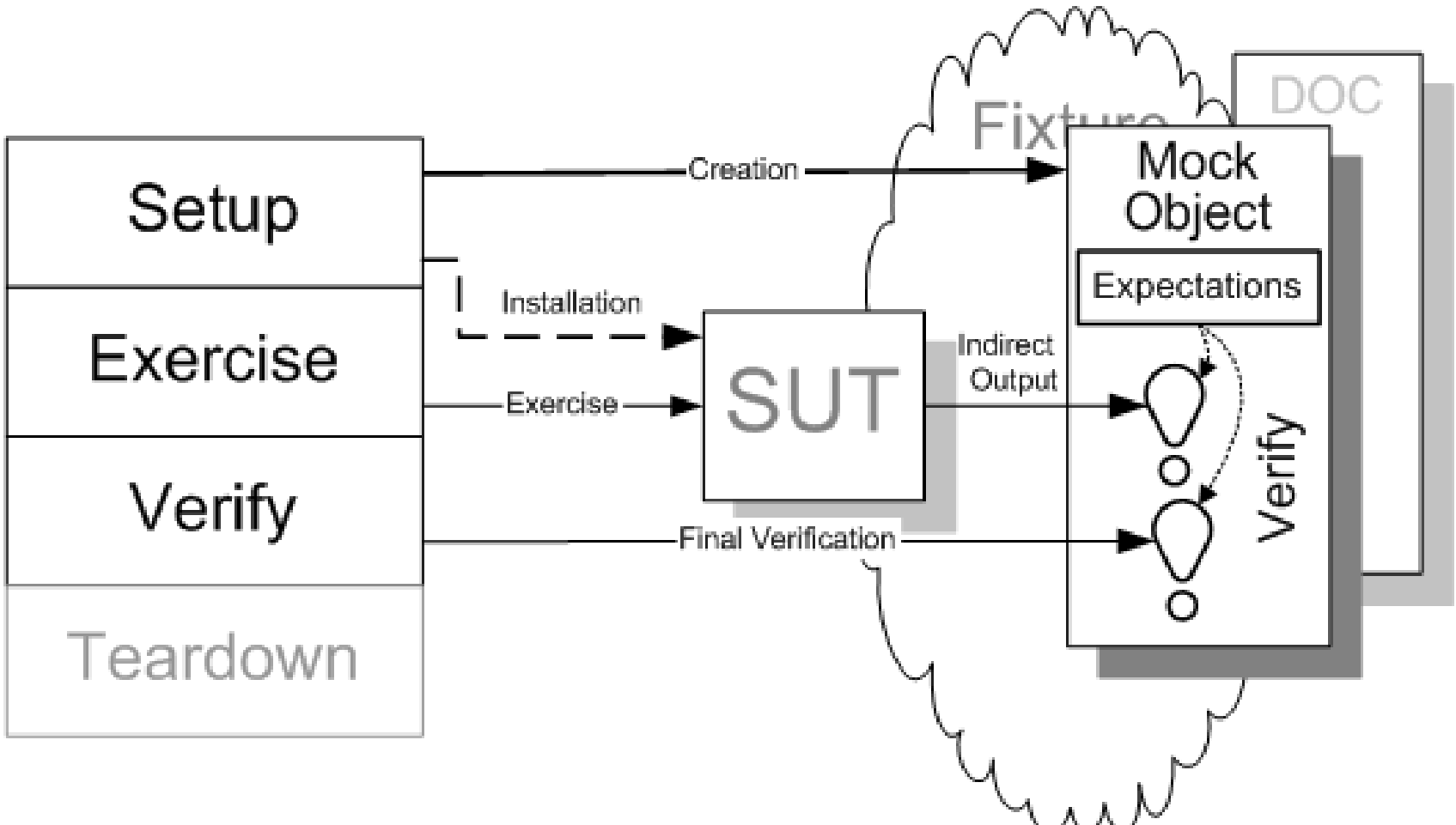
- We want to know SUT's interacts with one of its dependencies
- The spy only records the interaction, it is checked manually



Mock Object

RomanTranslatingTokenStream#testCorrectInputSingleOperator

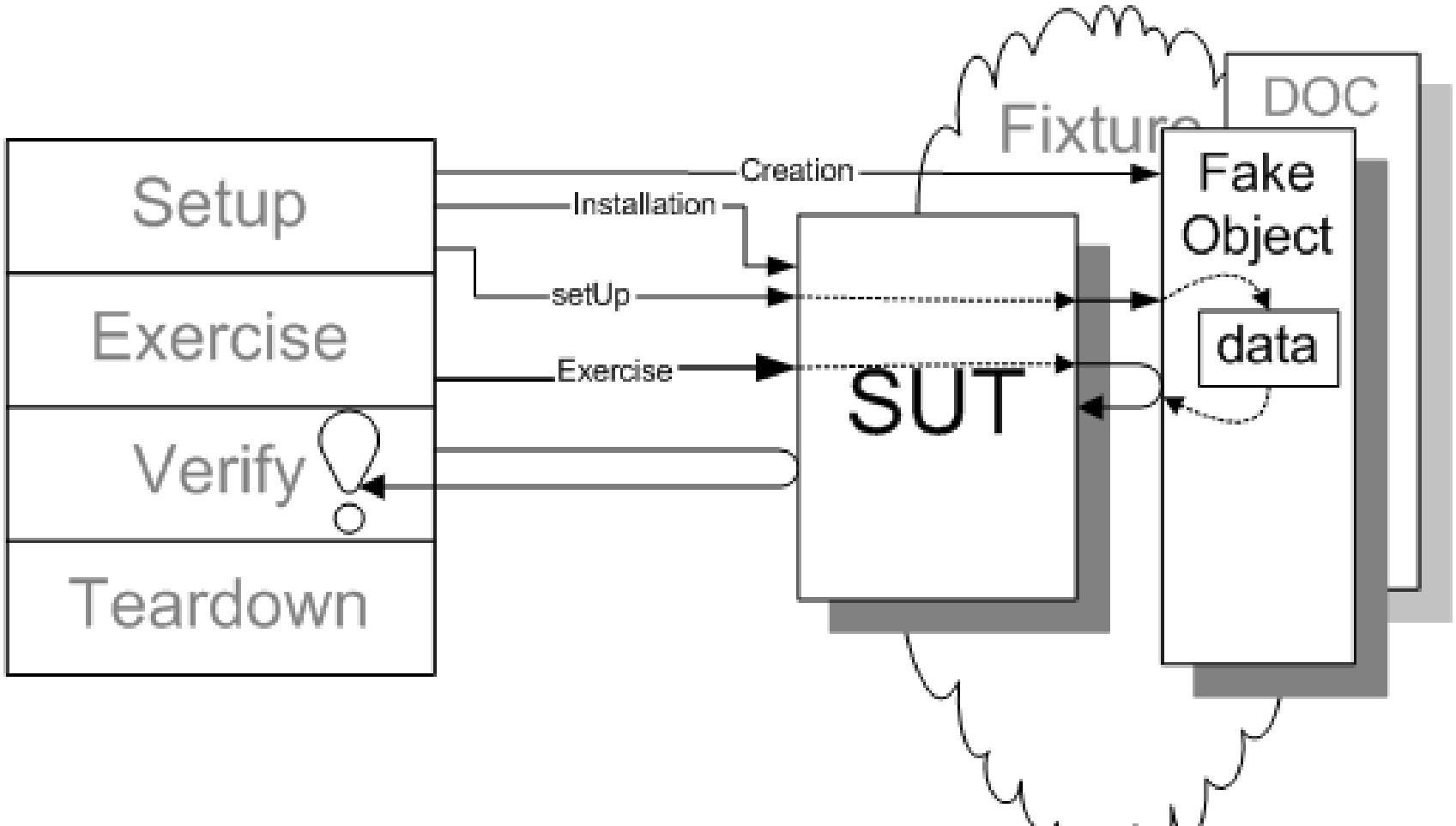
- Similar task as Test Spy, but checks the validity of SUT's interaction with the mock on the fly



Fake Object

No Example

- Has the same functionality as its real counterpart, but implements it in a more test friendly way
- e.g. an in-memory database instead of disk-based one



Task 1

- Write tests for the class *exercise/spy/ComplicatedClass*
- Test the following scenarios:
  - Input for the *doSomethingUseful* method is int lower or equalt to 10, thus nothing is logged
  - The input is 11 or more, thus there are two invocations on the logger, *setLevel* and *log*
  - As a bonus try to verify that the methods on the Logger are called in this exact order

Task 2

- Write tests for the class *exercise/stub/EmployeeManager*
- Test the following scenarios:
  - The Database provides valid result, at least two rows
  - The Database throws an exception
  - The Database returns some invalid data. These are thrown away, however all valid data are still processed