AARHUS UNIVERSITET

Microservices and DevOps

DevOps and Container Technology

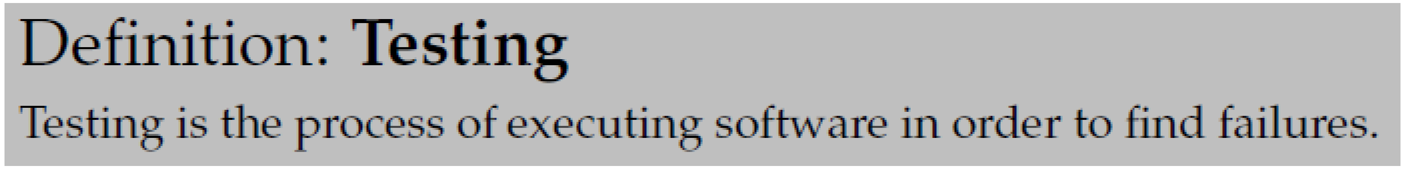
Test Doubles

Henrik Bærbak Christensen

AARHUS UNIVERSITET

Testability

- **Testability**: Concerned with the ease with which the software can be made to demonstrate its faults
- Techniques:
  - **Testing:**



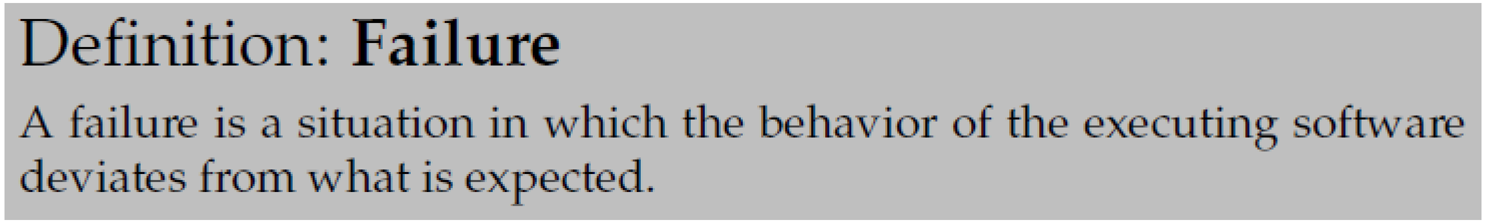
Definition: **Testing**

Testing is the process of executing software in order to find failures.

  - Review
    - Manual: Structured and systematic human *reading* of programs
    - Static analysis: let programs analyze your program
  - Formal verification: make profs that you program works

Failure and Defects

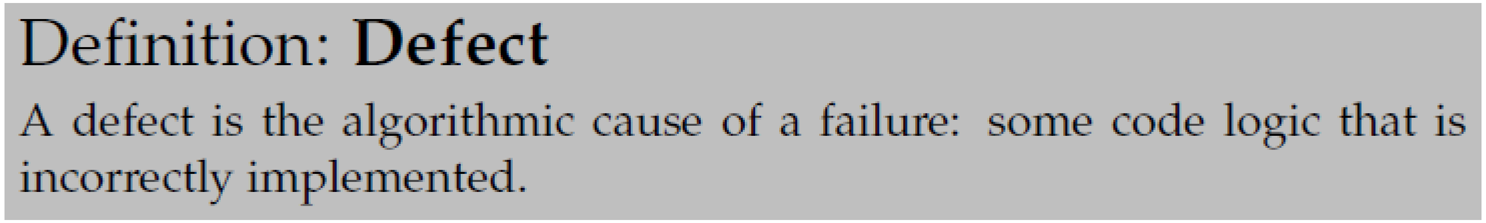
- What we observe when testing



Definition: **Failure**

A failure is a situation in which the behavior of the executing software deviates from what is expected.

- Why we observe it – the cause



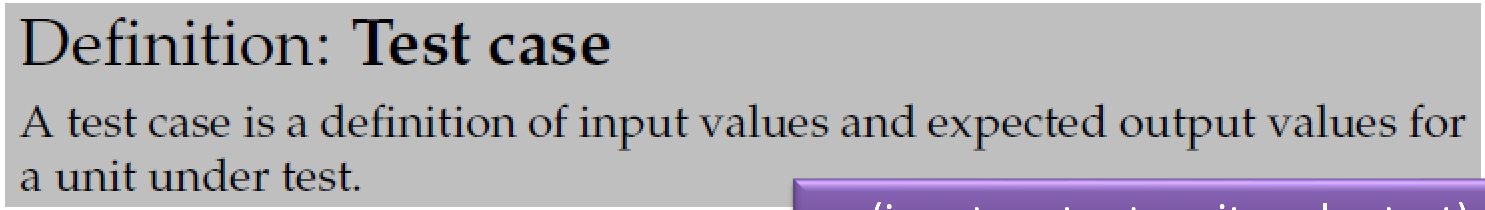
Definition: **Defect**

A defect is the algorithmic cause of a failure: some code logic that is incorrectly implemented.

- På dansk: Fejl og fejl ☺

Terminology

- Test Case



Definition: Test case

A test case is a definition of input values and expected output values for a unit under test.

(input, output, unit under test)

- Which means:
  - We have to isolate some part of the software – the *‘unit’*
  - We have to be able to *provide input* to the unit
  - We have to be able to *execute the unit with the input and observe the output (which requires a specific context)*
  - We have to know what *output to expect (oracle)*

AARHUS UNIVERSITET

Conclusion: Testing Issues

- Definition: **The Testability Input Issue**
  - Embody the issues involved in providing comprehensive and deterministic input to the unit under test in a reliable and reproducible way
- Definition: **The Testability Unit Isolation Issue**
  - Embody the issues involved in testing a unit under test in isolation in a comprehensive environment in a reliable and reproducible way
- Definition: **The Testability Output issue**
  - Embody the issues involved in recording the output from a unit under test and asserting the correctness in a reliable and reproducible way

AARHUS UNIVERSITET

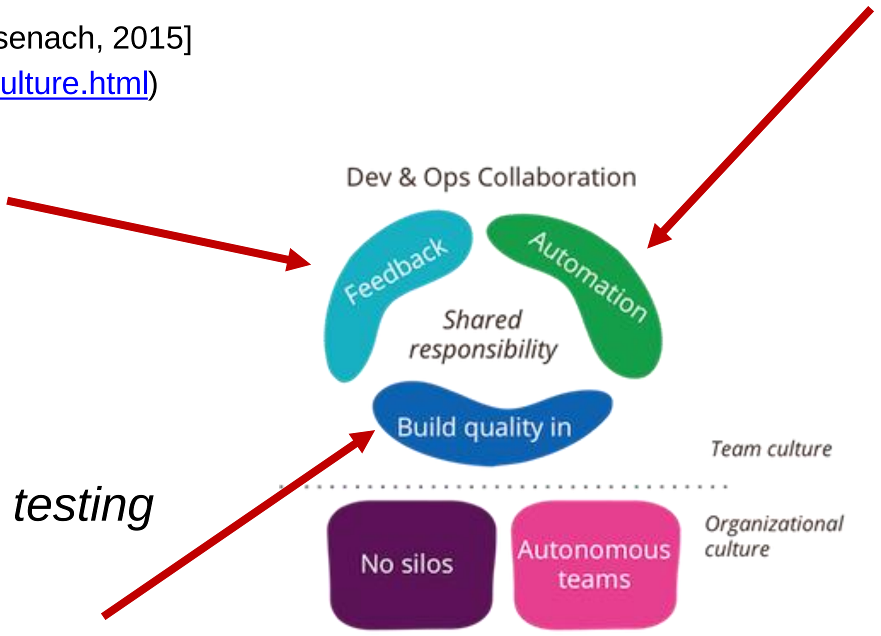
Testability and MSDO

I did not sign up for a test fagpakke, did I?

AARHUS UNIVERSITET

Yes you did ☺

- DevOps Culture [Rouan Wilsenach, 2015] (https://www.martinfowler.com/bliki/DevOpsCulture.html)
  - We need
    - Fast feedback
    - Quality Code
    - Automation
  - Main technique
    - *Automated regression testing*



So – in General

- All features/quality attributes should be *demonstrated through automated testing* in this course

- Write JUnit code to validate at unit testing level
  - Using test doubles to control *indirect input and ouput*

- Write JUnit+TestContainer code to validate at integration testing level
  - Use real-life containers to handle deterministic input and output
  - (And test double services or test doubles for non-determ.)

Test Doubles

Motivation

- Thorough testing requires software units to be tested in *isolation* – to create a ***test harness***/environment where defects/complexity in other units do not invalidate/complicate our testing.

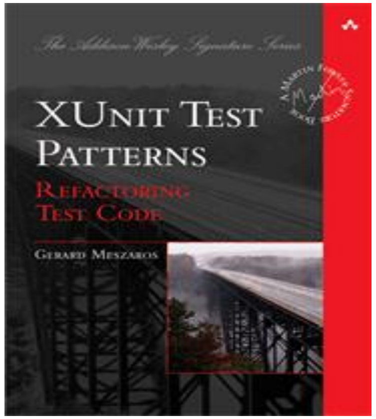
- The basic idea:
  - *Replace the unit(s) that the 'unit under test' collaborates with, with simpler and more controllable units*

AARHUS UNIVERSITET

Terminology

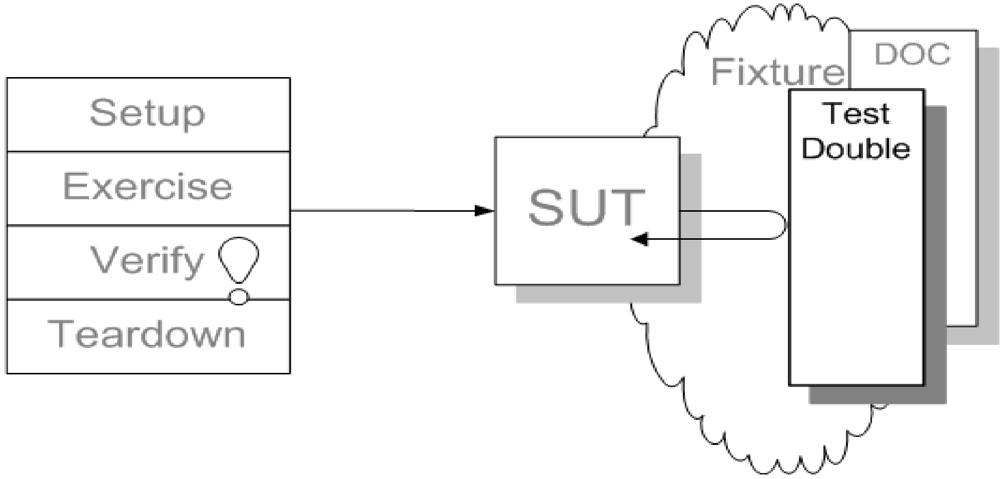
- These ‘replacement units’ have many names
  - Stubs, mocks, test drivers, skeletons, …
  - (I see a trend that many call everything ‘mocks’ because that is the fancy term ☺. But it is as *wrong*, as calling a banana for apple just because both are fruits…)

- Gerard Meszaros defines a clearer terminology by classifying the various uses of ‘replacements’...
  - Find it on www



xUnit Pattern: Test Double

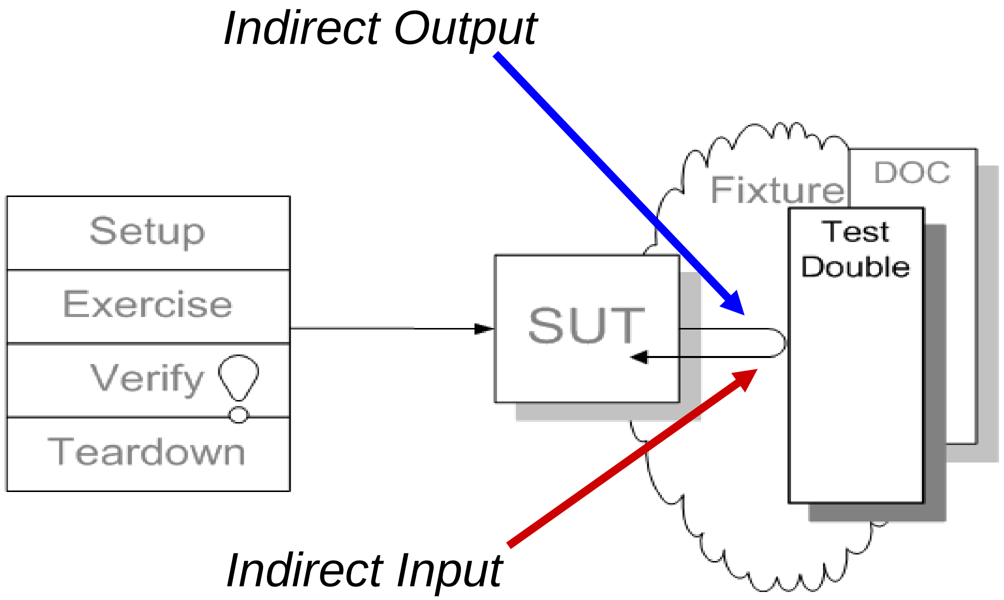
- "Superclass": **Test Double**
  - SUT: *System under test* (=UUT)
  - DOC: *Depended-on Component*
- When?
  - Slow tests
  - DOC is
    - not available
    - not under test control
    - has side-effects



AARHUS UNIVERSITET

Terminology

- *indirect output*
  - the output a UUT generates, not visible by our driver, but passed as parameters, protocols used, etc., to the DOCs
- *indirect input*
  - the input a Unit Under Test receives, not by parameter passing, instance variables, etc., but from results computed by DOCs.



AARHUS UNIVERSITET

Test Double

- Solution:
  - Replace DOC with a *double*
    - like stunt doubles in movies...
  - Requires:
    - *that this is possible!!!*

GoF's 1st principle:

Program to an interface...

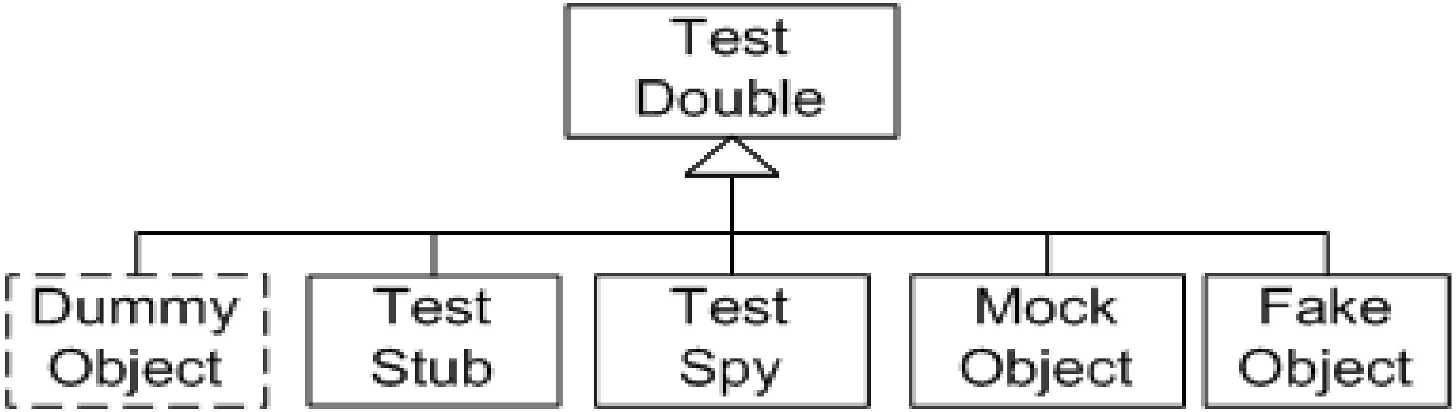
-

Dependency Injection!

AARHUS UNIVERSITET

Double classification

- Meszaros classify several types of doubles according to the specific testing perspective

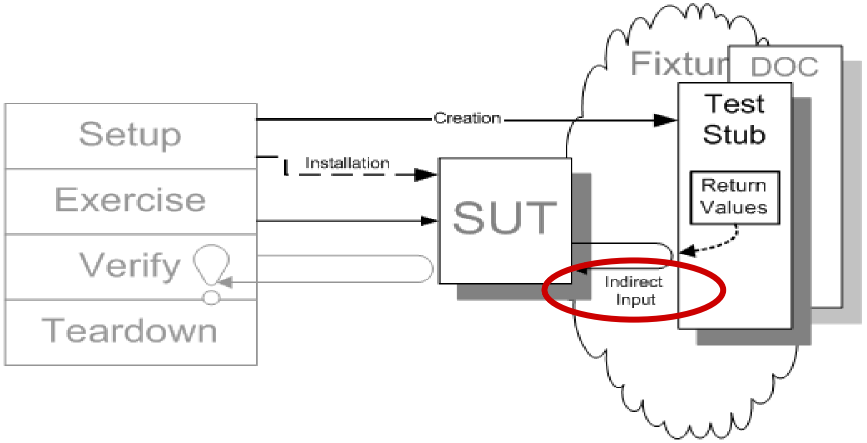


AARHUS UNIVERSITET

The Short Version

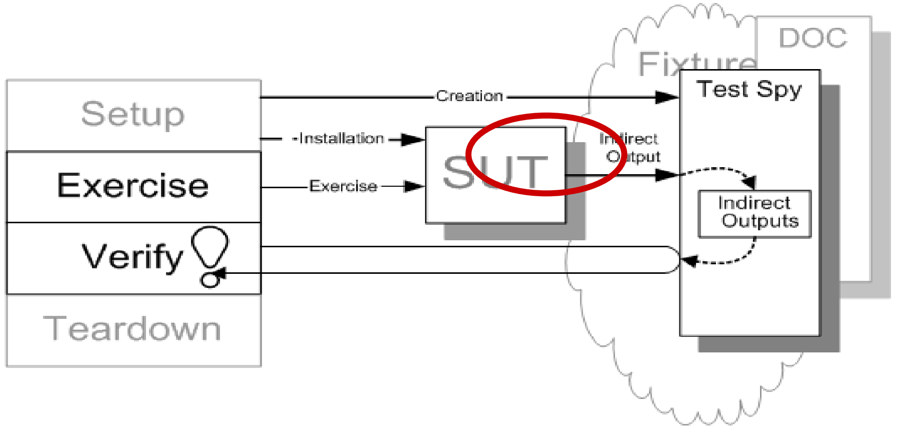
Test Stub

- Context
  - In many circumstances, the environment or context in which the system under test (SUT) operates very much influences the behavior of the SUT. To get good enough control over the indirect inputs of the SUT, we may have to replace some of the context with something we can control, a *Test Stub*.
- Example:
  - Test that the cooler starts when the temperature is 6 degrees
    - Stub the temperature sensor with a fixed return value of 6



Test Spy

- Context:
  - In many circumstances, the environment or context in which the SUT operates very much influences the behavior of the SUT. To get good enough visibility of the indirect outputs of the SUT, we may have to replace some of the context with something we can use to capture these outputs of the SUT.
- Example:
  - Test that the cooler starts when the temperature is 6 degrees
    - Replace cooling element with spy, that records when 'start()' is called; verify that it was called



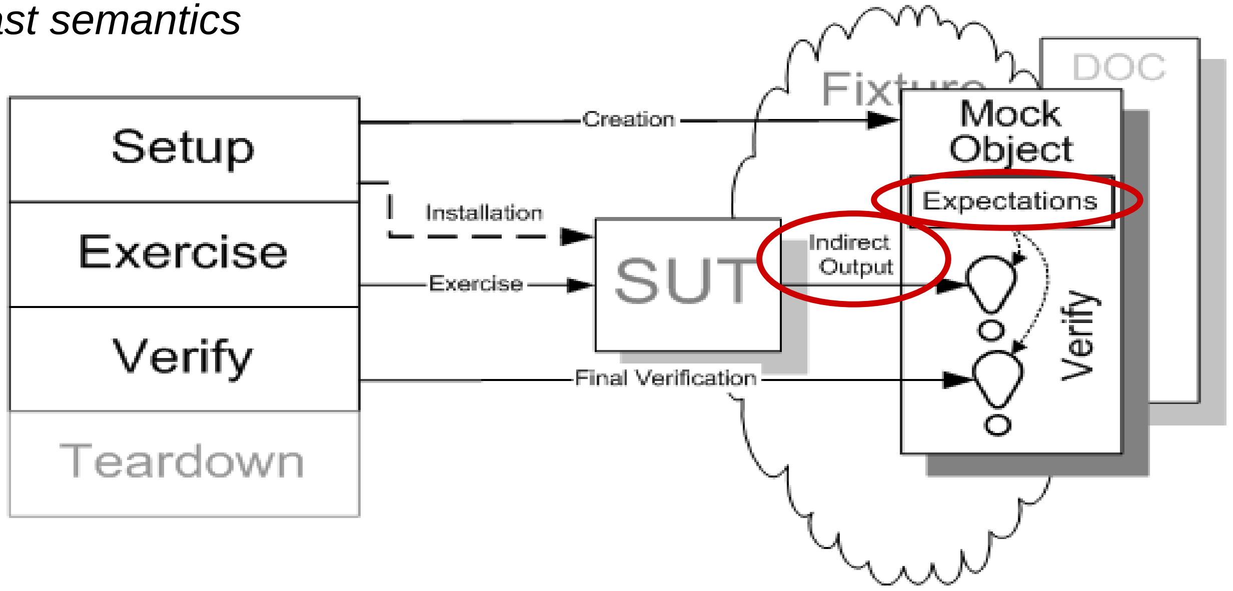
Combining

- Then our test case of the Cooling, c, may look like
  - Configure ‘stub = new TemperatureSensorStub(6);’
  - Configure ‘spy = new CoolingElementSpy();’
  - Dep inject into Cooler
    - cooler = new Cooler(stub, spy);
  - Execute test
    - cooler.regulateTemperature();   // read temp, if temp>6, start cooling
  - Validate that cooling element’s start method was called
    - assertThat(spy.lastInvokedMethod(), is(”start()”))

Mock Object

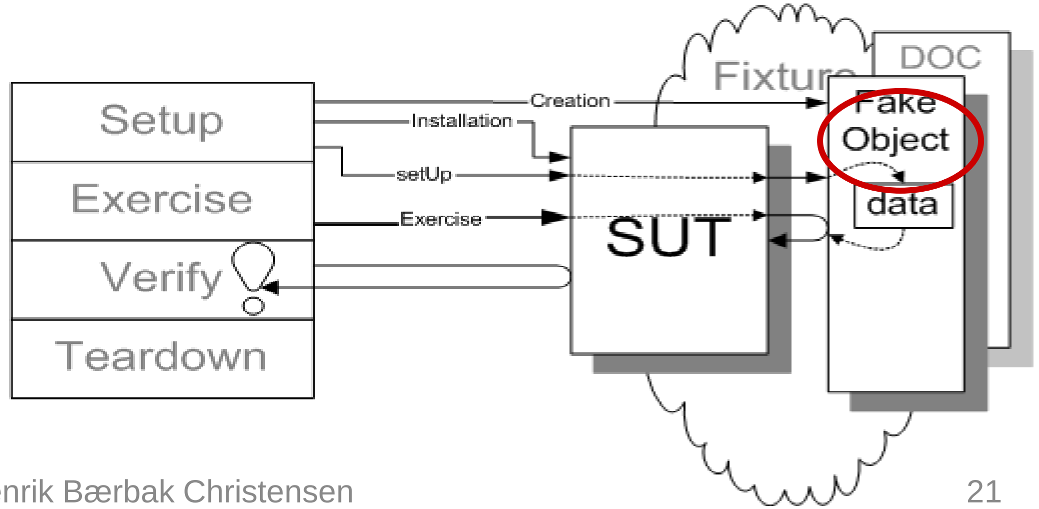
- Context:
  - A test double that verifies the indirect outputs
    - Usually *fail fast semantics*

- Use a mock library
  - Mockito, …



Fake Object

- Context:
  - The SUT often depend on other components or systems. The interactions with these other components may be necessary but the side-effects of these interactions as implemented by the real depended-on component (DOC), may be unnecessary or even detrimental. A *Fake Object* is a much simpler and lighter weight implementation of the functionality provided by the DOC without the side effects we choose to do without.



Microservice Context

- Meszaros’ terminology is founded in a ‘single system’ assumption
  - *I inject a FakeDB implementation of the Database interface, and test my Inventory implementation using that…*
- But the terminology is *independent* of the *connector* between the client and the server
  - In-process method call connector
  - Out-of-process REST call connector
- So – the stub can be
  - Java stub implementation
  - A remote service that provides stub values (ala Mountebank)

AARHUS UNIVERSITET

Summary

- **Test Doubles** allow you to get access to, inspect, and verify indirect input and output

- **Stub:** focus on indirect input
- **Spy:** focus on indirect output (record/verify)
- **Mock:** focus on indirect output (fail fast)
  - frameworks to generate doubles dynamically
- **Fake object:** light-weight semi-realistic behaviour

AARHUS UNIVERSITET

The Long Version

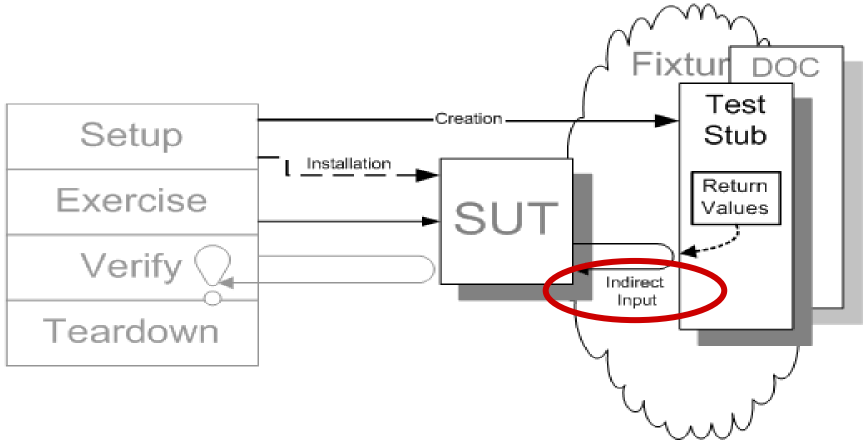
AARHUS UNIVERSITET

Test Stub

AARHUS UNIVERSITET

Test Stub

- Context
  - In many circumstances, the environment or context in which the system under test (SUT) operates very much influences the behavior of the SUT. To get good enough control over the indirect inputs of the SUT, we may have to replace some of the context with something we can control, a *Test Stub*.
- Examples:
  - ?



Test Stub Examples

- Typical examples are
  - Stubbing sensors or hardware
    - In a meteorological system it is important to test wind calculations over north when the wind direction changes from 359 degrees to 0 degrees.
  - Stubbing random behaviour
    - A dice must be put under test control

AARHUS UNIVERSITET

Stub variations

- *Responder*
  - used to inject **valid** indirect inputs: *happy paths*
- *Saboteur*
  - used to inject **invalid** indirect inputs
- *Temporary Test Stub*
  - a stand in for a *not-yet-implemented* DOC – the first TDD production code implementation is always of this kind.
    - this is what I called a *stub* in my book...
- *Entity Chain Snipping*
  - replace a network of objects with a single one

AARHUS UNIVERSITET

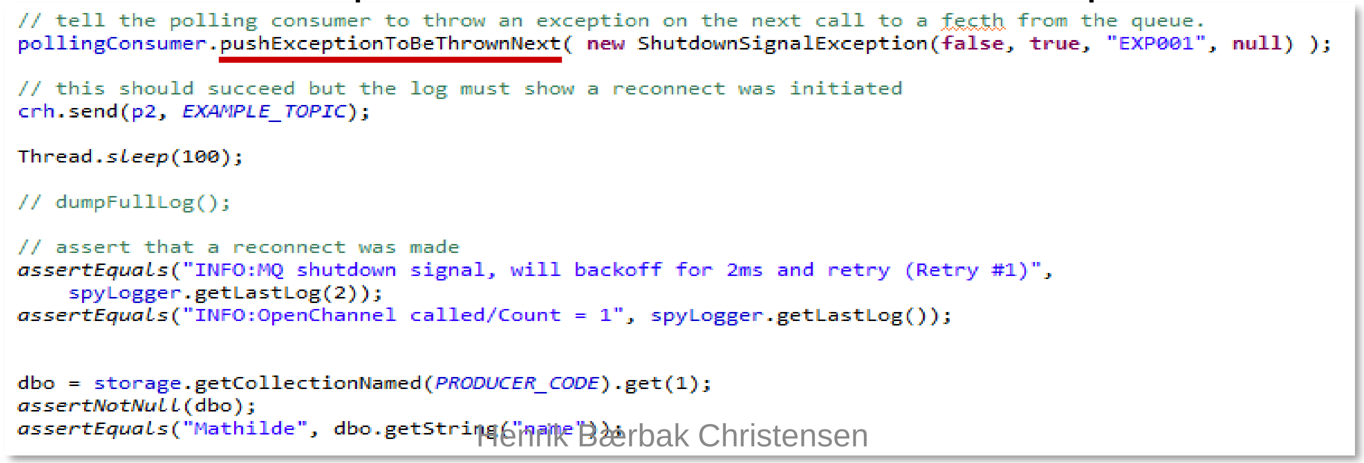
Example 1

- In Net4Care we generate XML documents representing telemedicine measurements
  - *Nancy has measured her weight to 77.0 kg*
- The format is PHMR, a document with a unique ID, generated at the time of creation
  - `<id root="2.16.840.1.113883.3.4208" extension="aa2386d0-79ea-11e3-981f-0800200c9a66"/>`
- However, comparing with ‘expected’ does not like random IDs ☹
- Solution
  - A UUIDStrategy interface, with a *responder* implementation

AARHUS UNIVERSITET

Example 2

- In EcoSense Karibu the daemons (responsible for fetching messages from our MessageQueue, converting them to JSON, and storing them in MongoDB), must react properly on MQ exceptions (ie. do a ”fail over”)
  - Introduce a *PollingConsumer* interface
    - A RabbitMQ implementation
    - A Saboteur implementation – that *will* throw exceptions



```
// tell the polling consumer to throw an exception on the next call to a fecth from the queue.
pollingConsumer.pushExceptionToBeThrownNext( new ShutdownSignalException(false, true, "EXP001", null) );

// this should succeed but the log must show a reconnect was initiated
crh.send(p2, EXAMPLE_TOPIC);

Thread.sleep(100);

// dumpFullLog();

// assert that a reconnect was made
assertEquals("INFO:MQ shutdown signal, will backoff for 2ms and retry (Retry #1)",
    spyLogger.getLastLog(2));
assertEquals("INFO:OpenChannel called/Count = 1", spyLogger.getLastLog());


dbo = storage.getCollectionNamed(PRODUCER_CODE).get(1);
assertNotNull(dbo);
assertEquals("Mathilde", dbo.getString("name"));
```

Test Stub

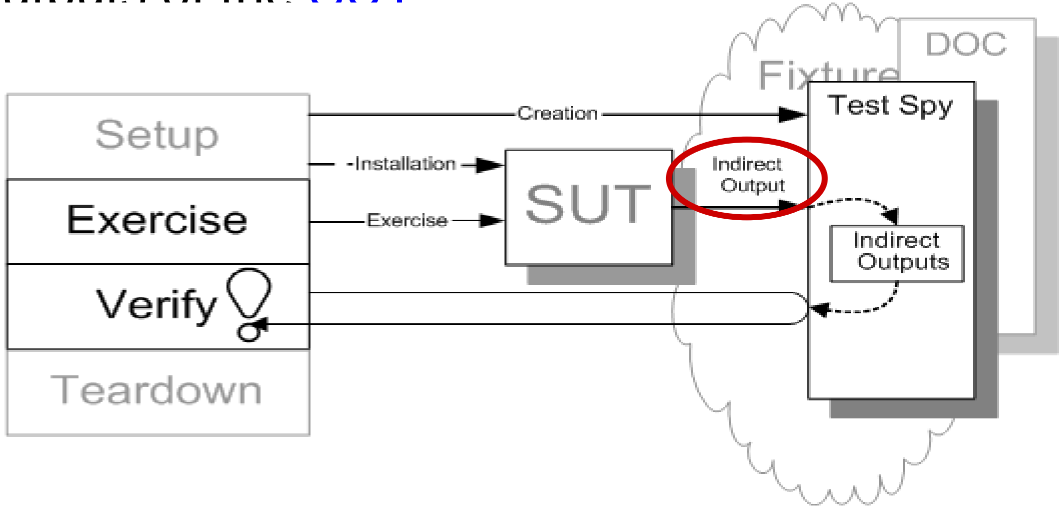
- Conclusion:
  - the primary purpose of the stub is
  - *to control the UUT’s input space*
- that is
  - we get *testing control over the input space + environment in order to specify test cases*
    - = (input, environment, expected output).
  - usually has methods/means for the test to *specify the returned indirect inputs*

Test Spy

AARHUS UNIVERSITET

Test Spy

- Context:
  - In many circumstances, the environment or context in which the SUT operates very much influences the behavior of the SUT. To get good enough visibility of the indirect outputs of the SUT, we may have to replace some of the context with something we can use to capture these outputs of the SUT



AARHUS UNIVERSITET

Test Spy Examples

- A Test Spy can
  - record the parameters passed to it
    - verify indirect computed output equals expected
  - the order in which DOC methods were called
    - verify the protocol between UUT and DOC

- A Test Spy does not fail, it merely records interaction.
- The Spy is inspected after the test execution in order to verify that indirect output was correct.

Implementation notes

- Test Spy inspection variations:
- *Retrieval interface:*
  - The spy must have additional methods to extract the stored indirect output
- *Self Shunt:*
  - The test case class itself implements the DOC interface and is thus feed the indirect output directly to be cached in local variables
- *Inner Test Double*
  - use an inner anonymous class as self shunt

Examples

- A classic example is an abstraction that communicates state changes via the Observer pattern
  - observer notification is an (important!) side-effect of state-changing method calls, but it is *not* externally visible: it is *indirect output*
  - register a SpyListener that counts the number of observer updates received.

Examples

- Gerry: An artificial Backgammon player
  - for all valid moves given board and dice
    - make move, compute value of board
    - if (value > bestvalue) { remember this move; }

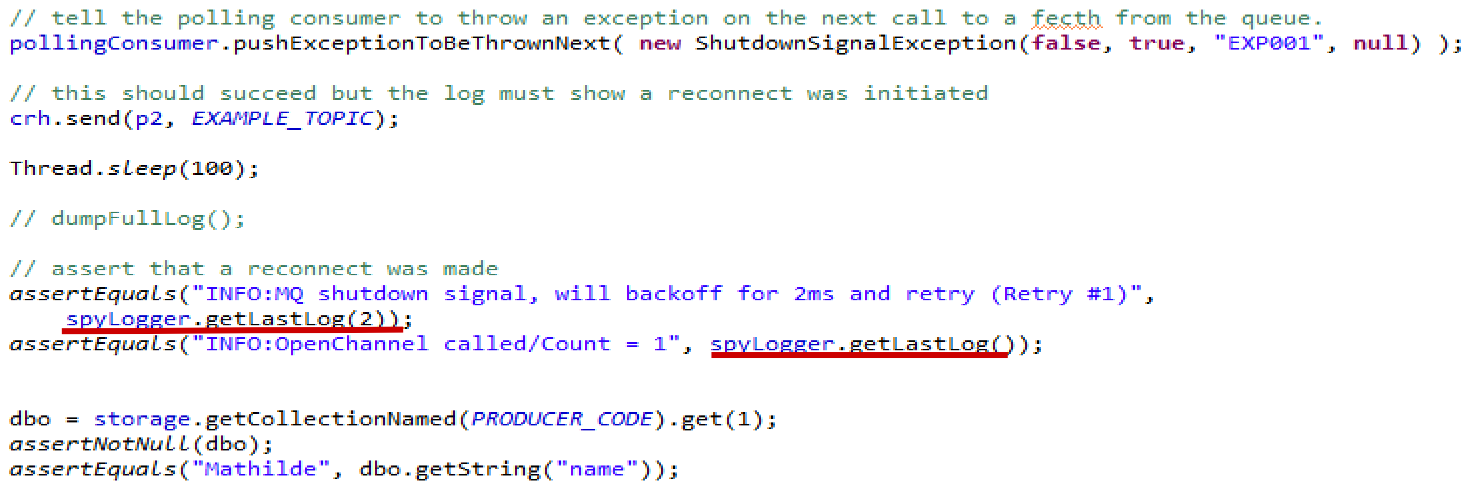
- But does it compute the proper moves? Is it really the best move that is taken?

- `movehook.considerMove(move);`
- normally considerMove is the empty method.

AARHUS UNIVERSITET

Examples

- EcoSense Karibu daemons of course log special situations like detected failures of nodes.
  - A SpyLogger is helpful as it
    - Verifies that log messages are indeed output
    - Verifies that failure situations are handled properly



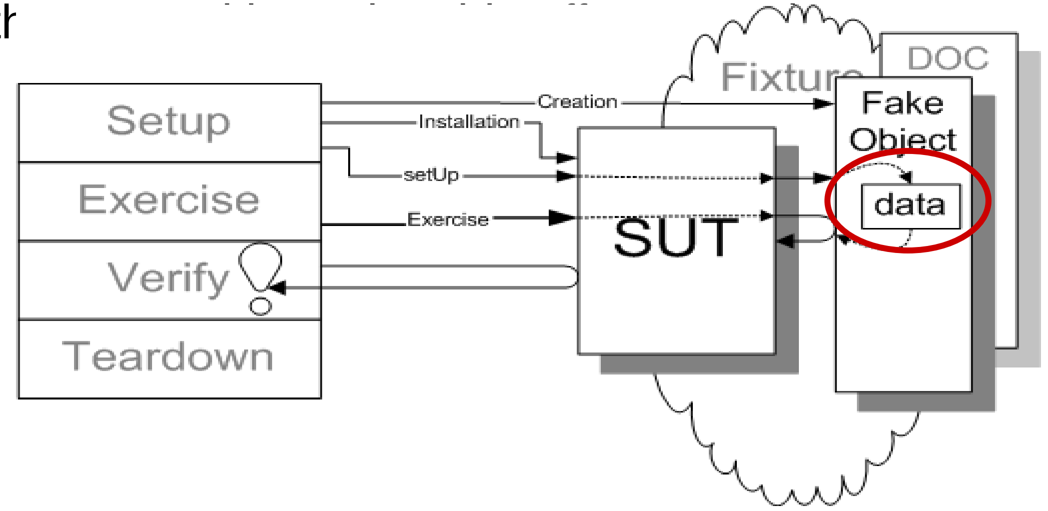
```
// tell the polling consumer to throw an exception on the next call to a fecth from the queue.
pollingConsumer.pushExceptionToBeThrownNext( new ShutdownSignalException(false, true, "EXP001", null) );

// this should succeed but the log must show a reconnect was initiated
crh.send(p2, EXAMPLE_TOPIC);

Thread.sleep(100);

// dumpFullLog();

// assert that a reconnect was made
assertEquals("INFO:MQ shutdown signal, will backoff for 2ms and retry (Retry #1)",
    spyLogger.getLastLog(2));
assertEquals("INFO:OpenChannel called/Count = 1", spyLogger.getLastLog());


dbo = storage.getCollectionNamed(PRODUCER_CODE).get(1);
assertNotNull(dbo);
assertEquals("Mathilde", dbo.getString("name"));
```

AARHUS UNIVERSITET

Fake Object

Fake Object

- Context:
  - The SUT often depend on other components or systems. The interactions with these other components may be necessary but the side-effects of these interactions as implemented by the real depended-on component (DOC), may be unnecessary or even detrimental. A *Fake Object* is a much simpler and lighter weight implementation of the functionality provided by tl
    do without.



Fake Object

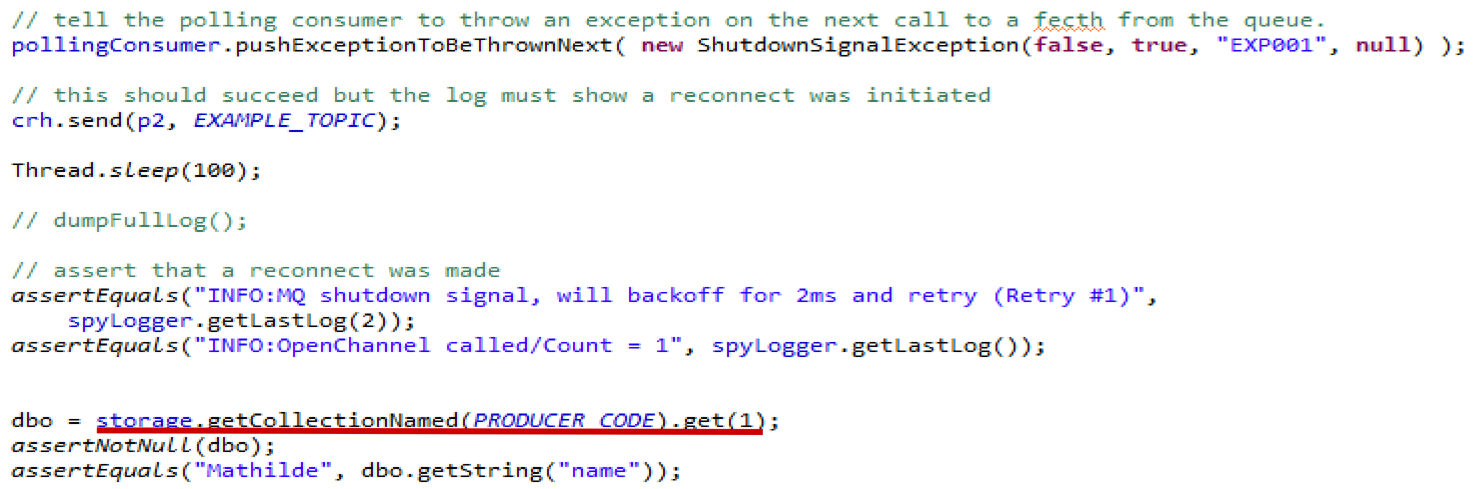
- Stub versus Fake Object?
  - Fake Object has “realistic” behaviour, a “lightweight version of the real implementation”
  - Fake Object is not instrumented/hard-coded with the indirect inputs, instead the indirect inputs comes from previous interactions with the SUT
    - Not “return 47”, but “return simpleDatastructure[index]”, whose contents is the result of previous interactions (store operations) with the SUT
  - Less focus on testing aspects of the SUT, more focus on making it work.

Types

- *Fake Database*
  - replace database with in-memory HashTables
- *In-Memory Database*
  - *semi-real database but not disk-based*
    - *(SQLite is brilliant in this respect ☺)*
- *Fake Web Service*
  - hard-coded or table-driven web server
- *Fake Service Layer*
  - fake the domain layer

Examples

- EcoSense Karibu daemons store documents in MongoDB but that is way to heavy for automated testing
  - StorageStrategy interface with *Fake Database*



```
// tell the polling consumer to throw an exception on the next call to a fecth from the queue.
pollingConsumer.pushExceptionToBeThrownNext( new ShutdownSignalException(false, true, "EXP001", null) );

// this should succeed but the log must show a reconnect was initiated
crh.send(p2, EXAMPLE_TOPIC);

Thread.sleep(100);

// dumpFullLog();

// assert that a reconnect was made
assertEquals("INFO:MQ shutdown signal, will backoff for 2ms and retry (Retry #1)",
    spyLogger.getLastLog(2));
assertEquals("INFO:OpenChannel called/Count = 1", spyLogger.getLastLog());


dbo = storage.getCollectionNamed(PRODUCER_CODE).get(1);
assertNotNull(dbo);
assertEquals("Mathilde", dbo.getString("name"));
```

AARHUS UNIVERSITET

Examples

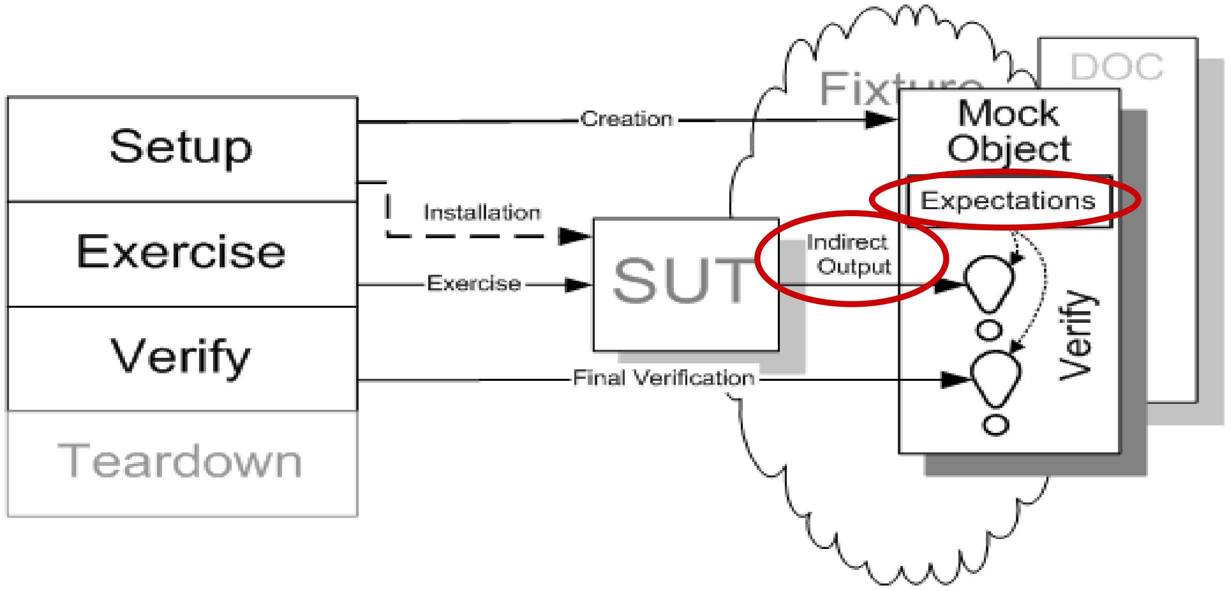
- Net4Care stores PHMR documents in *Cross-Enterprise Document Share* XDS.b repositories (SOAP web services) which is incredible heavy weight
  - XDS.b interface with a *In-Memory database (SQLite implementation)*
  - Allowed us to TDD the Net4Care framework
  - Allowed us to supply ‘local lightweight’ servers without the big overhead of getting real XDS.b server running

AARHUS UNIVERSITET

Mock Object

Mock Object

- Context:
  - A test double that verifies the indirect outputs



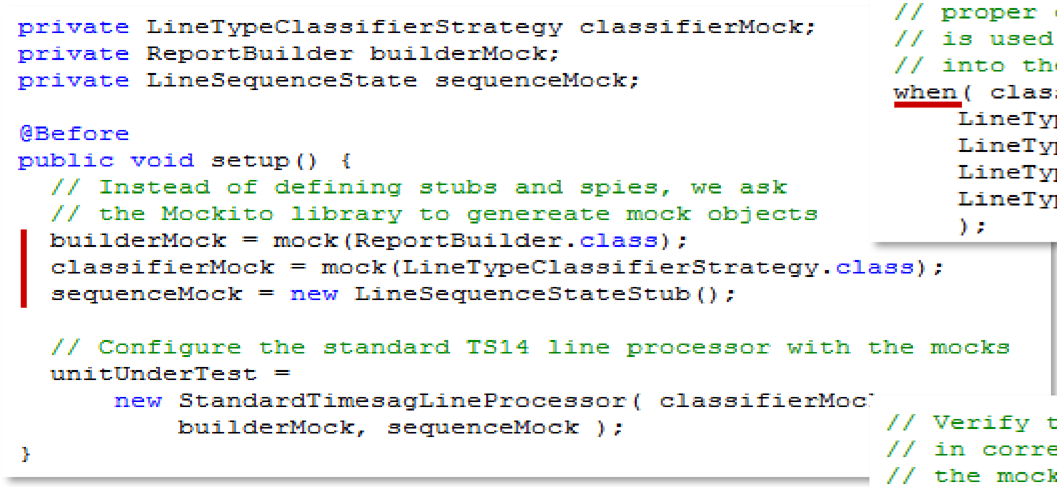
AARHUS UNIVERSITET

Mock Object Workings

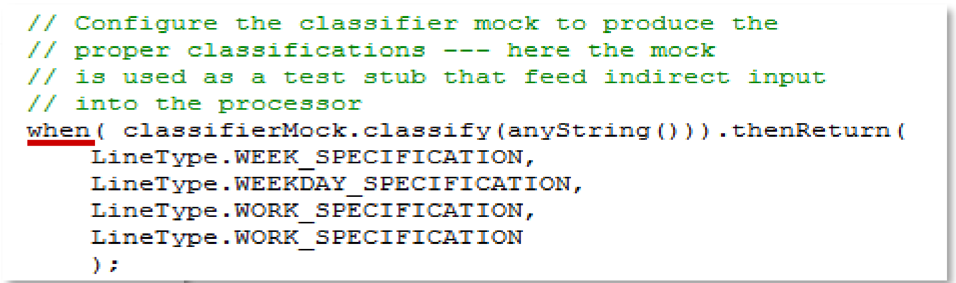
- Mock objects are somewhat more complex to define but are powerful to verify UUT behaviour with respect to the DOC.
  - Define Mock object with same interface as DOC
  - Configure mock with *expectations*
    - values to return (like test stub)
    - the methods that must be called
      - including sequence/protocol and call count
      - expected parameters
  - The mock will *fail* if these expectations are not met
    - fail fast!
  - Thus test driver needs not verify anything!

Mock Libraries

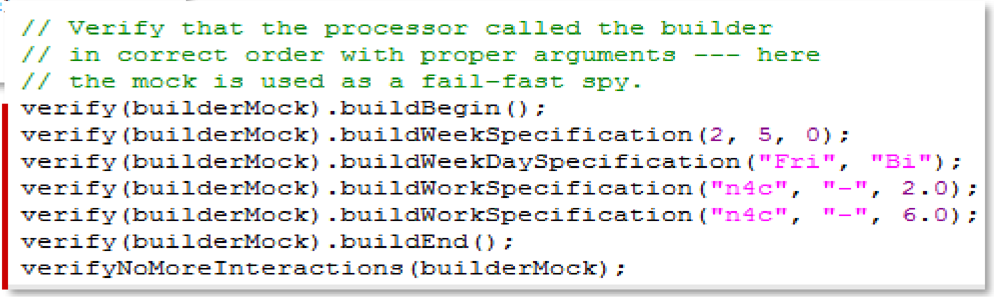
- Mocks are special in the sense that they are *dynamically created* by a Mock library



```java
private LineTypeClassifierStrategy classifierMock;
private ReportBuilder builderMock;
private LineSequenceState sequenceMock;

@Before
public void setup() {
  // Instead of defining stubs and spies, we ask
  // the Mockito library to genereate mock objects
  builderMock = mock(ReportBuilder.class);
  classifierMock = mock(LineTypeClassifierStrategy.class);
  sequenceMock = new LineSequenceStateStub();

  // Configure the standard TS14 line processor with the mocks
  unitUnderTest =
      new StandardTimesagLineProcessor( classifierMoc
          builderMock, sequenceMock );
}
```



```java
// Configure the classifier mock to produce the
// proper classifications --- here the mock
// is used as a test stub that feed indirect input
// into the processor
when( classifierMock.classify(anyString())).thenReturn(
        LineType.WEEK_SPECIFICATION,
        LineType.WEEKDAY_SPECIFICATION,
        LineType.WORK_SPECIFICATION,
        LineType.WORK_SPECIFICATION
        );
```



```java
// Verify that the processor called the builder
// in correct order with proper arguments --- here
// the mock is used as a fail-fast spy.
verify(builderMock).buildBegin();
verify(builderMock).buildWeekSpecification(2, 5, 0);
verify(builderMock).buildWeekDaySpecification("Fri", "Bi");
verify(builderMock).buildWorkSpecification("n4c", "-", 2.0);
verify(builderMock).buildWorkSpecification("n4c", "-", 6.0);
verify(builderMock).buildEnd();
verifyNoMoreInteractions(builderMock);
```

Considerations

- Mock objects must be programmed in advance
  - thus we must be able to predict UUT indirect output in advance – a hard-core whitebox requirement...
  - But pretty OK in a TDD context where you actually *program* the production code along with the test code.

- Mock objects are responsible for failing
  - thus exceptions thrown must be able to pass out of the UUT
    - may not be possible if it is embedded in a EJB container or similar...

MicroService Context

It is all so *in-memory* right?

AARHUS UNIVERSITET

Meszaros

- Meszaros’ terminology is founded in a ‘single system’ assumption
  - *I inject a FakeDB implementation of the Database interface, and test my Inventory implementaion using that…*
- But the terminology is independent of the *connector* between the client and the server
  - In-process method call connector
  - Out-of-process REST call connector
- So – the stub can be
  - Java stub implementation
  - A remote service that provides stub values (ala Mountebank)

AARHUS UNIVERSITET

Summary

- Testing units in *isolation* is important
  - unit and integration testing
  - test-driven development

- Units have more inputs and outputs than visible from the parameter list and instances variables
  - especially true in object-oriented programming
  - *indirect inputs*: data from DOCs
  - *indirect output:* data to DOCs

Summary

- **Test Doubles** allow you to get access to, inspect, and verify indirect input and output

- **Stub:** focus on indirect input
- **Spy:** focus on indirect output (record/verify)
- **Mock:** focus on indirect output (fail fast)
  - frameworks to generate doubles dynamically
- **Fake object:** light-weight semi-realistic behaviour