Introduction to testing with mock objects

Michael Pilquist

Agenda

- Testing Basics
- Mock Objects
- Java Tools
- Techniques
- Disadvantages and Pitfalls

Testing Basics

Testing Basics

- SUT = System Under Test
- Before writing a test, answer:
  - What's the SUT?
  - What's the goal of the test?

SUT = Entire application
Goal = Ensure all requirements are met

SUT = DefaultTaxCalculationStrategy class
Goal = Ensure taxes are calculated correctly

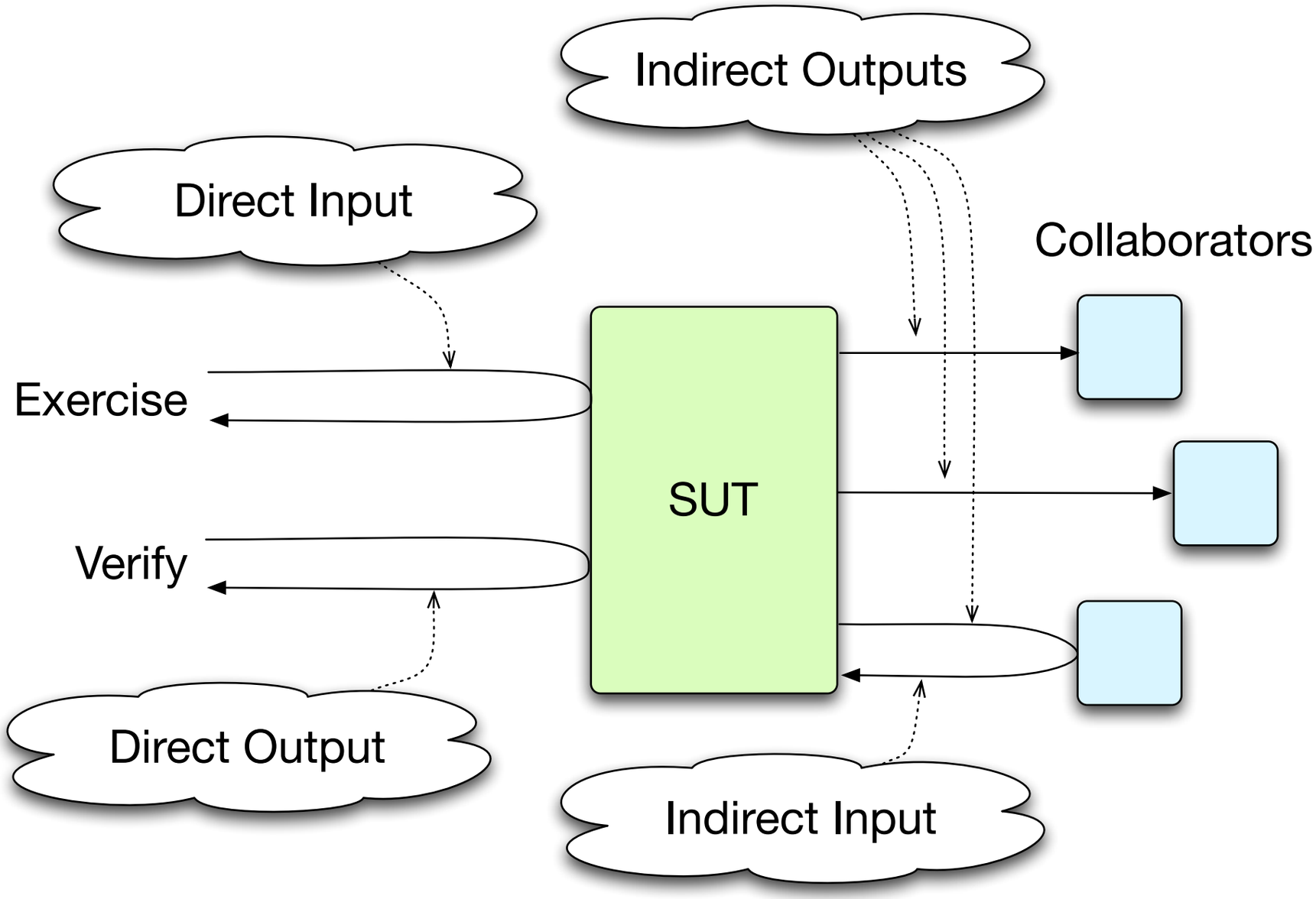
SUT = Persistence of Employee entity
Goal = Ensure entity can be persisted and hydrated via Hibernate

Basic Test Structure

- Set up
- Exercise
- Verify
- Tear down

```
@Test public class SessionManagerTest {
    private ExecutorService executor;
    private SessionManager manager;

    @BeforeMethod public void setUp() {
        executor = Executors.newCachedThreadPool();
        manager = new SessionManager(executor);
    }

    public void ensureNoSessionsExistAfterInitialization() {
        Assert.assertTrue(manager.getSessions().isEmpty());
    }

    @AfterMethod public void tearDown() {
        executor.shutdownNow();
    }
}
```

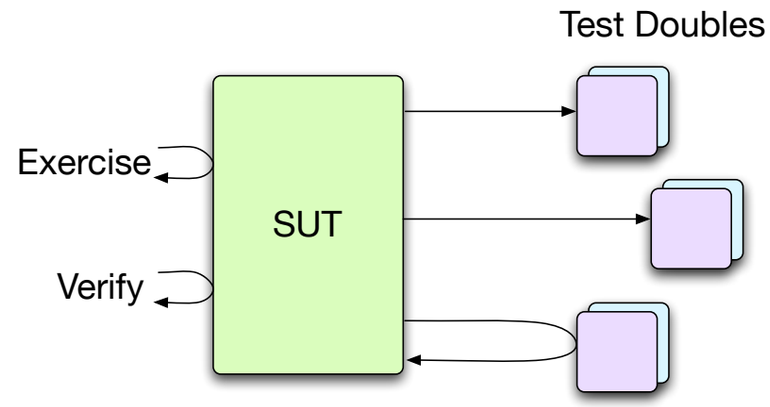
Control and observation points



See “xUnit Test Patterns” by Gerard Meszaros

Testing in isolation

- Isolate the SUT from its dependencies
- Replace objects outside the SUT with **test doubles**



- Utilize inversion of control to ease installation of test doubles

- Allows testing of code paths that may not be easily reproducible with the real objects
- Allows testing of code that would be slow if using the real objects
- Allows testing of code that does not provide the necessary observation points

Types of test doubles

- Gerard Meszaros's taxonomy:
  - Test stub - test double used to control the indirect inputs of the SUT
  - Test spy - test double that captures the indirect outputs of the SUT for later verification
  - Mock object - test double that verifies the indirect outputs of the SUT as the SUT is exercised
  - Fake object - test double that is not used to control the indirect inputs or observe the indirect outputs; e.g., simple implementation of the real object that doesn't have side effects
  - Dummy object - test double that is used in place of a real object when neither the test nor the SUT is interested in the object

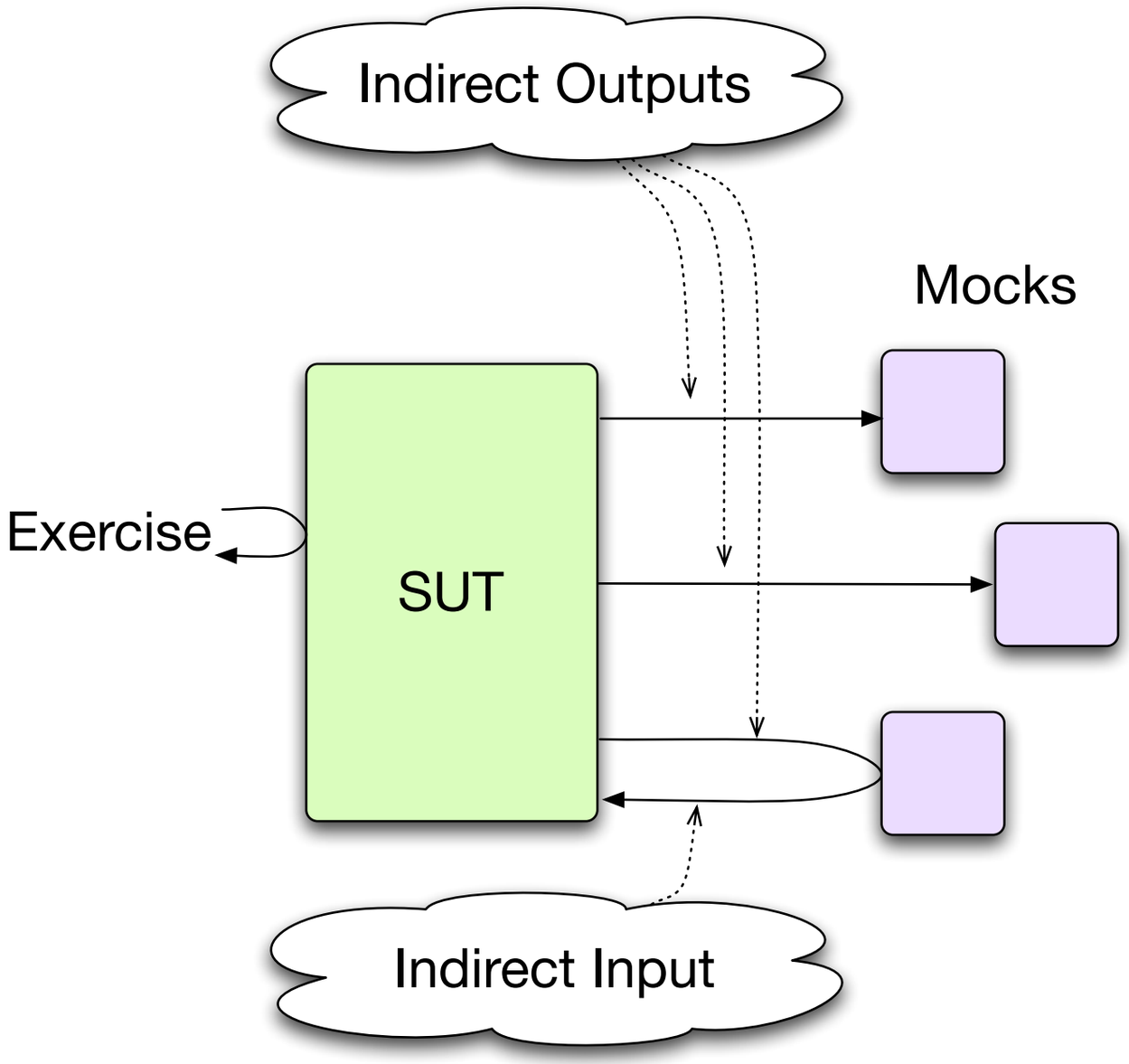
See "xUnit Test Patterns" by Gerard Meszaros

Mock Objects

- Mock object = test double that verifies the **indirect output** of the SUT

- Indirect outputs of the SUT = "*actions that cannot be observed through the public application programming interface of the SUT but that are seen or experienced by other systems or application components*" --Meszaros

- One style of behavior verification



See "xUnit Test Patterns" by Gerard Meszaros

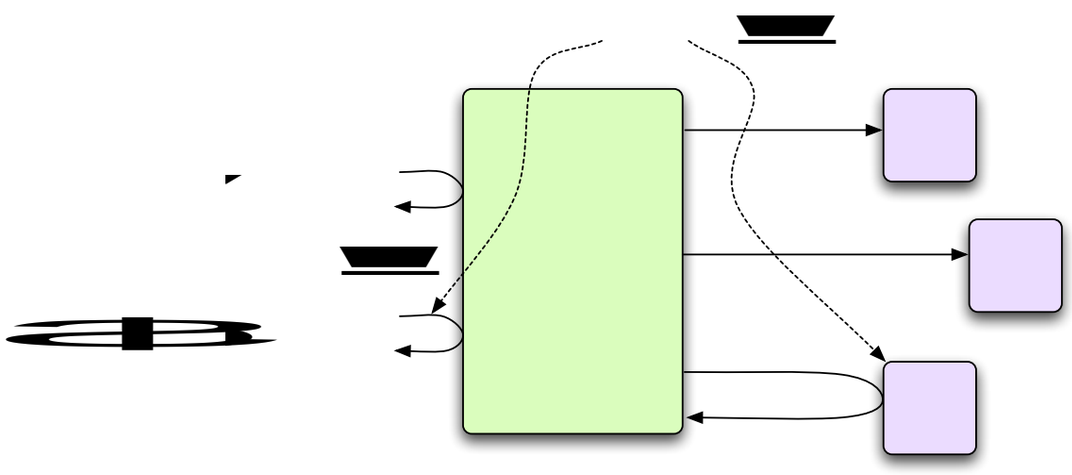
Test Structure Using Mock Objects

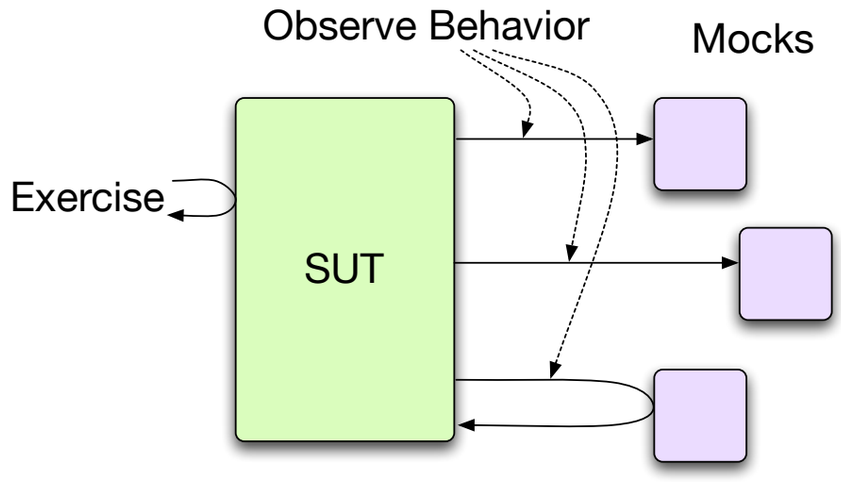
- Set up
- Exercise
- Verify
- Tear down

```
@Test public class SessionManagerTest {
    private ExecutorService executor;
    private SessionManager manager;

    @BeforeMethod public void setUp() {
        executor = EasyMock.createMock(ExecutorService.class);
        manager = new SessionManager(executor);
    }

    public void ensureNoSessionsExistAfterInitialization() {
        EasyMock.replay(executor);
        Assert.assertTrue(manager.getSessions().isEmpty());
        EasyMock.verify(executor);
    }

    @AfterMethod public void tearDown() {
        // Nothing to do
    }
}
```

State vs. Behavior

State	Behavior
Verify state of SUT after exercise	Verify behavior of SUT during or after exercise
SUT typically is a black box	SUT is necessarily a white box
Easier to read and write	Harder to read and write (possibly due to unfamiliarity)





State vs. Behavior: Which should I use?

- Not an either/or
  - Some tests clearly require state verification
  - Some tests clearly require behavior verification

- Martin Fowler defines two approaches to deciding when to use mocks
  - Classical TDD - *“use real objects if possible and a double if it’s awkward to use the real thing”*
  - Mockist TDD - *“always use a mock for any object with interesting behavior”*

See “Mocks Aren’t Stubs” by Martin Fowler

Java Tools

EasyMock

- http://www.easymock.org/
- Generates mock implementations of interfaces at runtime
- Supports mocking abstract & concrete classes
- Recording metaphor
  - Mock objects start in recording mode
  - Expectations are expressed by calling methods on the mock objects
  - After all expectations are expressed, mocks are switched in to verification mode

Simple EasyMock Example

```
public class EasyMockWarehouseTest {
    private Warehouse warehouse;

    @BeforeMethod public void setUp() {
        warehouse = new DefaultWarehouse();
    }

    @Test public void ensureOrderReceivedEventIsDispatchedWhenAnOrderIsServiced() {
        final WarehouseEventListener listener =
            EasyMock.createMock(WarehouseEventListener.class);
        warehouse.addListener(listener);

        listener.orderReceived((WarehouseEvent) EasyMock.notNull());
        EasyMock.replay(listener);

        final Order order = new Order();
        warehouse.serviceOrder(order);

        EasyMock.verify(listener);
    }
}
```

Simple EasyMock Example

```
FAILED: ensureOrderReceivedEventIsDispatchedWhenAnOrderIsServiced
java.lang.AssertionError:
  Expectation failure on verify:
    orderReceived(notNull()): expected: 1, actual: 0
     at org.easymock.internal.MocksControl.verify(MocksControl.java:82)
     at org.easymock.EasyMock.verify(EasyMock.java:1410)
     at ete2008.EasyMockWarehouseTest.ensureOrderReceivedEventIsDispatchedWhenAnOrderIs
Serviced(EasyMockWarehouseTest.java:24)
... Removed 22 stack frames
```

jMock

- http://www.jmock.org/
- Generates mock implementations of interfaces at runtime
- Supports mocking abstract & concrete classes
- DSL for expressing expectations
  - Mockery is the central object
  - Mocks are created through the mockery
  - Expectations are configured through the mockery

Simple jMock Example

```
public class JMockWarehouseTest {
    private Warehouse warehouse;

    @BeforeMethod public void setUp() {
        warehouse = new DefaultWarehouse();
    }

    @Test public void ensureOrderReceivedEventIsDispatchedWhenAnOrderIsServiced() {
        final Mockery context = new Mockery();

        final WarehouseEventListener listener =
            context.mock(WarehouseEventListener.class);
        warehouse.addListener(listener);

        context.checking(new Expectations() {{
            one(listener).orderReceived(with(aNonNull(WarehouseEvent.class)));
        }});

        final Order order = new Order();
        warehouse.serviceOrder(order);

        context.assertIsSatisfied();
    }
}
```

Simple jMock Example

```
FAILED: ensureOrderReceivedEventIsDispatchedWhenAnOrderIsServiced
not all expectations were satisfied
expectations:
  expected exactly 1 time, never invoked: warehouseEventListener.orderReceived(not
null); returns a default value
    at org.jmock.Mockery.assertIsSatisfied(Mockery.java:190)
    at ete2008.JMockWarehouseTest.ensureOrderReceivedEventIsDispatchedWhenAnOrderIs
Serviced(JMockWarehouseTest.java:28)
... Removed 22 stack frames
```

Techniques

Actions

- Many times it is necessary to specify how a mock should respond to a method call

- If a method on a mock returns a value, the return value must be specified

```
EasyMock.expect(orderRepository.save(order)).andReturn(order);
EasyMock.expect(orderRepository.save(order)).andThrow(new DataAccessException());
EasyMock.expect(orderRepository.getAll()).andReturn(Arrays.asList(order));
```

- More complicated actions can be handled

```
EasyMock.expect(orderRepository.save(order)).andAnswer(new IAnswer<Order> {
    public Order answer() throws Throwable {
        final Order originalOrder = (Order) EasyMock.getCurrentArguments()[0];
        final Order savedOrder = new Order(originalOrder);
        savedOrder.setId(SAVED_ID);
        return savedOrder;
    }
});
```

Matchers

- Example test cases earlier were **underspecified**

```
listener.orderReceived((WarehouseEvent) EasyMock.notNull());
```

  - Event source and order attributes could be wrong

- Two solutions

  - Override equals and hashCode and change expectation to:

```
listener.orderReceived(new WarehouseEvent(warehouse, order));
```

  - Implement a custom **matcher**

```
class WarehouseEventMatcher implements IArgumentMatcher {
    private final Warehouse expectedSource;
    private final Order expectedOrder;

    public WarehouseEventMatcher(final Warehouse source, final Order order) {
        expectedSource = source;
        expectedOrder = order;
    }

    @Override public boolean matches(final Object argument) {
        if (!(argument instanceof WarehouseEvent))
            return false;

        final WarehouseEvent event = (WarehouseEvent)argument;
        final boolean sourceMatches = expectedSource == null
            ? event.getSource() == null : expectedSource.equals(event.getSource());
        final boolean orderMatches = expectedOrder == null
            ? event.getOrder() == null : expectedOrder.equals(event.getOrder());

        return sourceMatches && orderMatches;
    }

    @Override public void appendTo(final StringBuffer buffer) {
        buffer.append(
            String.format("warehouseEvent(%s, %s)", expectedSource, expectedOrder));
    }
}
```

```
private static WarehouseEvent warehouseEvent(
    final Warehouse warehouse, final Order order) {
    EasyMock.reportMatcher(new WarehouseEventMatcher(warehouse, order));
    return null;
}

@Test public void ensureOrderReceivedEventIsDispatchedWhenAnOrderIsServiced() {
    final WarehouseEventListener listener =
        EasyMock.createMock(WarehouseEventListener.class);
    warehouse.addListener(listener);
    final Order order = new Order();

    listener.orderReceived(warehouseEvent(warehouse, order));
    EasyMock.replay(listener);

    warehouse.serviceOrder(order);

    EasyMock.verify(listener);
}
```

Matchers

- EasyMock comes with many matchers

```
someMock.someMethod(and(geq(10), leq(20));
someMock.otherMethod(startsWith(“/prefix”));
```

- jMock comes with a set of basic matchers but is designed to be used with Hamcrest

```
one(otherMock).someMethod(with(hasEntry(key, value)));
one(otherMock).otherMethod(with(anyOf(is(“java”), is(“scala”), is(“groovy”))));
```

- Hamcrest can also be used with EasyMock

- Hamcrest can also be used to write test assertions in JUnit 3.x, JUnit 4, and TestNG

```
assertThat(sut.getLanguages(), hasItems(“haskell”, “ocaml”, “sml”));
```

Ordering

- By default, EasyMock and jMock do not enforce the order of methods
- Order of methods is sometimes meaningful to describe the correct behavior of the SUT

> Order must be persisted in the repository before the order received event is dispatched to event listeners

Ordering in EasyMock

- EasyMock supports a *strict mock*
  - Ordering of method calls on a strict mock is enforced
  - But this does not handle ordering of methods across a group of mocks
- EasyMock supports a *strict control*
  - Mocks are created by the strict control
  - Strict control enforces ordering

```
OrderRepository repository =
    createStrictMock(OrderRepository.class);
expect(repository.save(order))
    .andReturn(order);
expect(repository.getById(10L))
    .andReturn(order);
replay(repository);
...
verify(repository);
```

```
IMocksControl control = createStrictControl();
OrderRepository repository =
    control.createMock(OrderRepository.class);
WarehouseEventListener listener = control.
    createMock(WarehouseEventListener.class);

expect(repository.save(order)).andReturn(order);
listener.orderReceived(order);

control.replay();
...
control.verify();
```

Ordering in jMock

- jMock's approach is based on defining a sequence of behavior
  - Specify which expectations are sequenced
  - More flexible than EasyMock's approach
    - A mock can take part in any number of sequences
    - A single expectation can take part in any number of sequences

```
final Sequence serviceOrder =
    context.sequence("service order");
context.checking(new Expectations() {{
    one(orderRepository).save(order);
    returnValue(order);
    inSequence(serviceOrder);

    one(listener).orderReceived(
        with(aNonNull(WarehouseEvent.class)));
    inSequence(serviceOrder);
}});
```

Expecting behavior multiple times

- What if a method should be called N times?

```
EasyMock.expect(orderRepository.getAll()).andReturn(orders).times(2);
exactly(2).of(orderRepository).getAll(); returnValue(orders);
```

- What if a method *may* be called but doesn't *have* to be called?

```
EasyMock.expect(orderRepository.getAll().andReturn(orders).anyTimes();
allowing(orderRepository).getAll(); returnValue(orders);
```

- jMock proposes a rule of thumb to decide when to allow behavior vs. expect behavior
  - *"Allow queries and expect commands"*

Creating stubs or other test doubles

- EasyMock and jMock can create stubs and other test doubles as well as mocks (i.e., test doubles that don't verify behavior)

- Useful when you need an implementation of an interface but don't want or can't use a real implementation

```
private Employee createStubEmployee(String name) {
    final Employee employee = EasyMock.createNiceMock(Employee.class);
    EasyMock.expect(employee.getName()).andStubReturn(name);
    EasyMock.replay(employee);
    return employee;
}
```

- A nice mock allows any methods to be called any number of times. If the method returns a value, a zero value is returned (0, null, false)

Disadvantages and Pitfalls

Refactoring Difficulties

- Minor refactorings to implementation details can break tests
    - Leads to ignored test failures / commented out tests
- Rename method refactoring works with current versions of EasyMock and jMock
- Larger refactorings can have large impacts
- Ways to avoid
    - Watch for overspecified tests

See “Mock Roles, Not Objects” by Steve Freeman, Nat Pryce, Tim Mackinnon, and Joe Walnes

Overspecified Tests

- Too many expectations or too specific expectations lead to tests that fail due to the slightest of refactorings
- Many times, expectations are added just to get a test to pass
- Ways to avoid
  - Keep principles of behavior verification in mind when writing expectations
  - Consider using stubs instead of mocks if the expectations do not help meet the goal of the test

See “xUnit Test Patterns” by Gerard Meszaros

Underspecified Tests

- Expectations are written in such a general way that no useful behavior is being tested, but the test passes and the coverage percentage is high

```
expect(orderRepository.save((Order)anyObject())).anyTimes();
```

- Ways to avoid
  - Keep in mind that test coverage percentages do not equate to well tested systems
  - Clearly identify the goal of a test; if verifying the interaction of the SUT with its collaborators isn't part of the goal, consider replacing the mocks with stubs

See "xUnit Test Patterns" by Gerard Meszaros

False Security

- Tests utilizing mocks and other test doubles pass but when integrated in production, failures occur, despite high test coverage

- Due to unexpected/untested interactions

- Ways to avoid
  - Complement unit test suite with functional test suite

See “Next Generation Java Testing” by Cedric Beust and Hani Suleiman

Mock Overload

- Tests become very difficult to read due to excessive use of mocks and expectations

- Ways to avoid
  - Only write expectations for behaviors that contribute to meeting the goal of the test
  - May be a design issue with the SUT
    - Check for compliance with the Single Responsibility Principle

Mocking Complicated Third Party APIs

- Tests are difficult to read and modify due to mocking of a complicated third party API
- Many of the benefits of TDD with mocks do not apply since the third party API cannot be changed
- Mock objects often do not behave the same as the third party API
- Ways to avoid
  - Look for test support libraries that provide test doubles (e.g., Spring's mock implementations of the servlet API)

See "Next Generation Java Testing" by Cedric Beust and Hani Suleiman

Summary

- Mock objects are an effective way to implement tests that utilize behavior verification
- Mock objects are not a silver bullet -- just one part of a comprehensive testing strategy
- Investigate EasyMock, jMock, or other mocking frameworks
- Consider using the mocking frameworks to make stub generation easier

Resources

“Mocks Aren’t Stubs” by Martin Fowler
http://martinfowler.com/articles/mocksArentStubs.html

“xUnit Test Patterns: Refactoring Test Code” by Gerard Meszaros

“Mock Roles, Not Objects” by Steve Freeman, Nat Pryce, Tim Mackinnon, and Joe Walnes
http://www.jmock.org/oopsla2004.pdf

http://mockobjects.com/

“Next Generation Java Testing: TestNG and Advanced Concepts” by Cedric Beust and Hani Suleiman