Génie Logiciel Avancé
Cours 6 — Testing Object Oriented Systems

Mo Foughali
foughali@irif.fr

Laboratoire IRIF, Université Paris Cité

2021–2022

Outline

Outline

OO design and messaging

> *The big idea is "messaging" [...] The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.*
> — *Alan Kay*

Intuition

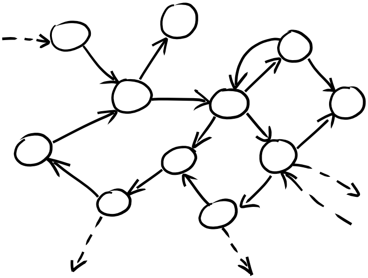
- invoke method `m` on object `obj` $\sim$ send message `m` *to* object `obj`

Upon reception of message `m`:

- `obj` can react, sending other messages to his neighbors
- `obj` can respond, returning a value or raising an exception

A web of objects

- the behavior of an OO system is an emergent property of object composition
- corollary: an OO system should be organized as:
  1. a set of composable objects
  2. a declarative description of how to compose them
     - e.g., in the program's `main`, or in a configuration file
     - by (only) changing object composition, you can change the behavior of the system



GOOS, Figure 2.1

Some objects are more equal than others

For design and testing purposes, we distinguish:

values (or "functional objects") model immutable entities that do not change over time. Values have no identity, i.e., for the purposes of the system there is no significant difference between *different* objects that encode the same information

- in Java, we usually compare values with `.equals()`

objects (or "computational objects") model stateful entities, whose state change over time, and model computational processes (e.g., algorithms, local behavior, etc). Different computational objects with—at present—the same state have different identities and cannot be exchanged, because their states can diverge in the future (e.g., if they receive different messages)

- in Java, we usually compare objects with ==

Protocols as interfaces

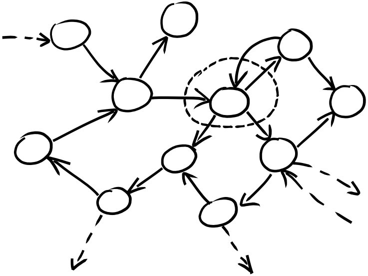
To easily change the global behavior of an OO system...
you need to be able to easily replace objects...
and to achieve that you need:

- explicit dependencies between objects
- establish common communication protocols
  - our "interfaces" are no longer limited to static parameter/return value typing, but now span dynamic object behavior

Result: all objects that follow the same protocol are mutually interchangeable, once instantiated on the same dependencies.

This is a significant mental shift from the static classification of objects as instances of classes organized in a single hierarchy. Usually you can have a single class hierarchy; but you can have many different protocols, with multi-faceted classifications.

Unit-testing collaborating objects



GOOS, Figure 2.4

/o\

We have an OO system and we want to unit-test an object (the encircled one).

- we want the test to be isolated
  - failures in *other* objects shouldn't affect *this* object's unit tests
- testing method I/O is not enough
- we need to test adherence to the expected communication protocol
  - does it send to its neighbor the expected messages?
  - in the right order?
  - does it respond appropriately?
- we have to do so without knowledge about its internal state ("tell, don't ask")

Example — testing observer

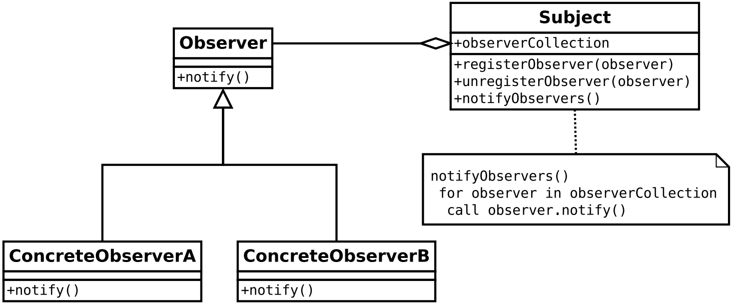
Observer design pattern — reminder

The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. —

https://en.wikipedia.org/wiki/Observer_pattern

We want to unit test a Java implementation of the observer design pattern.

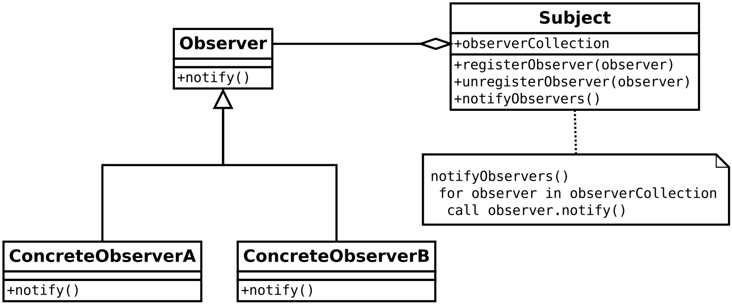
Observer — what to test



https://en.wikipedia.org/wiki/File:Observer.svg

what should we test?

Observer — what to test

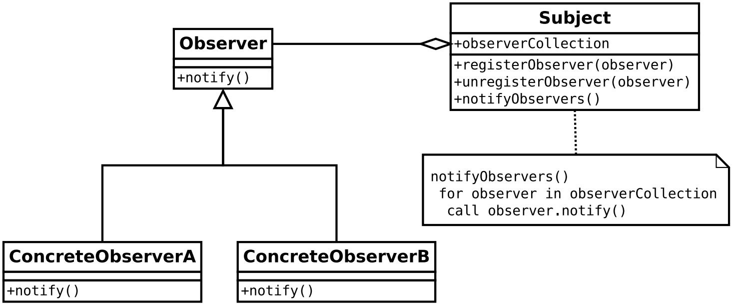


https://en.wikipedia.org/wiki/File:Observer.svg

what should we test?

(many things, among which) that adding observers “works”

Observer — what to test



https://en.wikipedia.org/wiki/File:Observer.svg

what should we test?

(many things, among which) that adding observers "works", i.e.:

- ~~registerObserver does not throw an exception~~ (not enough)
- ~~registerObserver returns nothing~~ (?!?!)
- ~~register/unregister round trip~~ (how w/o internal state?)
- ...

Observer — what to test (cont.)

adding observers "works", i.e.:

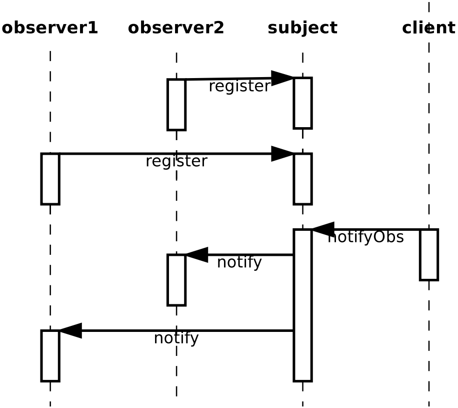


Figure – 1 subject with 2 observers, sequence diagram

Observer — what to test (cont.)

adding observers "works", i.e.:

- upon notification `notify` is called on all registered observers
- upon registration `notify` is *not* called
- registering twice results in double notifications
- ...

i.e., that our subject implements the expected protocol

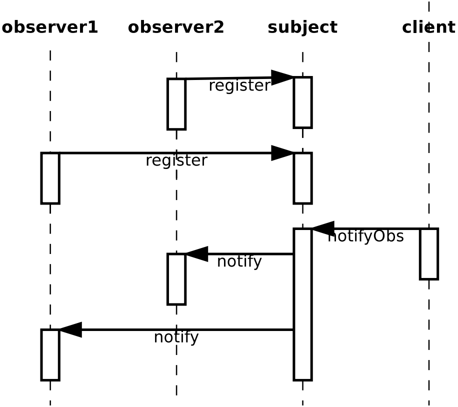


Figure – 1 subject with 2 observers, sequence diagram

Observer — what to test (cont.)

adding observers “works”, i.e.:

- upon notification `notify` is called on all registered observers
- upon registration `notify` is *not* called
- registering twice results in double notifications
- ...

i.e., that our subject implements the expected protocol

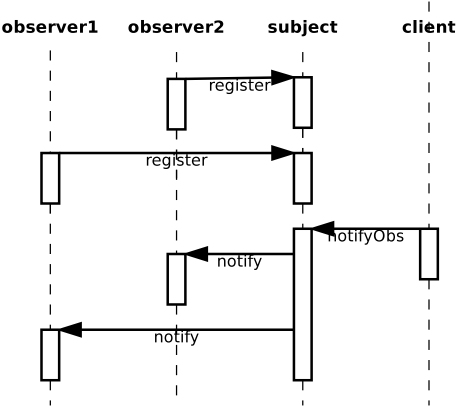


Figure – 1 subject with 2 observers, sequence diagram

let’s try testing this with JUnit...

Example — interfaces

```java
public interface Observer {
    void notify (String message);
}

public interface Observable {
    void addObserver(Observer o);
    void notifyObservers(String msg);
}
```

Example — implementation

```java
public class TrivialSubject implements Observable {
    private ArrayList<Observer> observers =
        new ArrayList<Observer>();

    public void addObserver(Observer o) {
        observers.add(o);
    }

    public void notifyObservers(String msg) {
        for (Observer o: observers) { o.notify(msg); }
    }
}

public class StdoutObserver implements Observer {
    public void notify(String message) {
        System.out.println(message);
    }
}
```

Example — JUnit test, single observer

Test: trivial subject notifies single observer once upon notify

```
public class TrivialSubjectTestJUnit {
    private TrivialSubject subj = new TrivialSubject();

    @Test
    public void notifiesSingleObserverOnceUponNotify() {
        Observer obs = new Observer() {
            public void notify(String msg) {
                throw new RuntimeException();
            }
        };
        subj.addObserver(obs);
        try {
            subj.notifyObservers("triviality");
            fail("subject did not call notify");
        } catch (RuntimeException e) {
            // do nothing, this is the expected behavior
        }
    }
```

Example — JUnit test, single observer (cont.)

Slightly more readable syntax for expectations, but the logic is still convoluted:

```
public class TrivialSubjectTestJUnit {
    private TrivialSubject subj = new TrivialSubject();

    @Test(expected = RuntimeException.class)
    public void notifiesSingleObserverOnceUponNotify() {
        Observer obs = new Observer() {
            public void notify(String msg) {
                throw new RuntimeException();
            }
        };
        subj.addObserver(obs);
        subj.notifyObservers("triviality");
    }
}
```

Trivia: are we actually checking that notify is called *once*?

Example — JUnit test, double observer

Test: trivial subject notifies twice a double observer upon notify

```
private int notifications = 0;

private void bumpNotificationCount() {
    notifications++;
}

@Test public void
notifiesDoubleObserverTwiceUponNotify() {
    Observer obs = new Observer() {
        public void notify(String msg) {
            bumpNotificationCount();
        }
    };
    subj.addObserver(obs);
    subj.addObserver(obs);
    subj.notifyObservers("triviality");
    assertEquals(2, notifications);
}
```

Example — JUnit test, double observer (cont.)

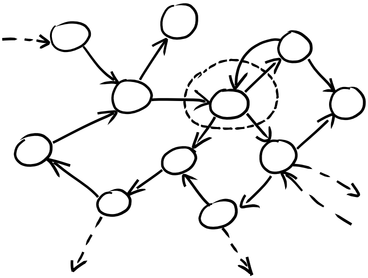
Discussion:

- quite a bit of gymnastic to track the actual notification count
  - mostly Java-specific: anonymous classes can't easily affect the surrounding context
- readability: test is now arguably obscure; at a glance one might ask:
  - where does `notifications` come from?
  - is it cleaned-up between tests?
- the purpose of this test is very similar to the previous one (1 vs 2 notifications), but the test code looks very different

Problems with this approach

More generally, there are at least 3 classes of problems with using xUnit to test object protocols:

- to test an object, we instantiate concrete classes for its neighbors
  - creating "real" objects might be difficult; we can only mitigate with builders
  - we sacrifice isolation: neighbors' bugs might induce test failures



GOOS, Figure 2.4

- we piggyback expectations onto complex mechanisms
  - e.g., scaffolding to count invocations...
- we lack an expressive language for protocol expectations, e.g.:
  - check that `notify` invocation count belongs to an interval
  - expectations on arguments/return values
  - expectations on invocation ordering
  - ...

Outline

Object mocking

Object mocking is a technique used to address all these problems.

Idea

To test an object we replace its neighbors with "fake" objects—called mock objects—which are easier to create than concrete objects.

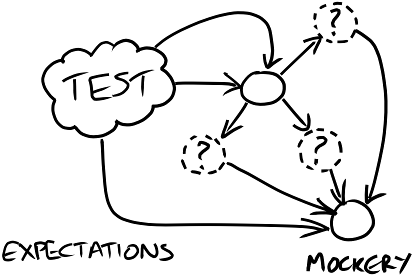
We then define expectations on how the object under test should behave w.r.t. mock objects, i.e., which messages the tested object exchanges with mock objects.

GOOS, Figure 2.5

Mock scaffolding

In analogy with xUnit, mock frameworks support mocking offering:

- mockery: objects that hold test context, create mock objects, and manage expectations
- expressive DSL to define readable expectations



GOOS, Figure 2.6

Mock test structure:

1. create required mock objects
2. create the object under test
3. define expectations on how mock objects will be called
4. call the triggering method
5. assert result validity (and fulfillment of expectations)

Example — single observer test, with mocks

Test: trivial subject notifies single observer once upon notify [1]

```java
private TrivialSubject subj = new TrivialSubject();
@Rule
public JUnitRuleMockery context = new JUnitRuleMockery();

@Test
public void notifiesSingleObserverOnceUponNotify() {
    final Observer obs = context.mock(Observer.class);
    final String msg = "triviality";
    subj.addObserver(obs);
    context.checking(new Expectations () {{
        oneOf (obs).notify(msg);
    }});
    subj.notifyObservers(msg);
    // no assertions
    // expectations implicitly verified by jMock
}
```

1. using JUnit+jMock, we'll discuss details later

Example — double observer test, with mocks

Test: trivial subject notifies single observer once upon notify

```
@Test
public void notifiesDoubleObserverTwiceUponNotify() {
    final Observer obs = context.mock(Observer.class);
    final String msg = "triviality";
    subj.addObserver(obs);
    subj.addObserver(obs);
    context.checking(new Expectations () {{
        exactly(2).of (obs).notify(msg);
    }});
    subj.notifyObservers(msg);
}
```

Differences w.r.t. previous test (highlighted in the code):

- extra `addObserver` call
- `oneOf` → `exactly(2).of`

jMock

As it happens for xUnit, there exist several mock object frameworks, for different platforms and languages.

In these slides we use jMock (2.x), originally by Steve Freeman and Nat Pryce, who also popularized object mocking with the book *Growing Object-Oriented Software, Guided by Tests*.

jMock is a popular mock framework for Java, which integrates with JUnit's test runner and assertion engine (for extra validation on top of expectations).

- homepage: `http://www.jmock.org/`
- jMock is Free Software, BSD-licensed

jMock — Hello, world!

```java
import static org.junit.Assert.*;
import org.junit.Rule;
import org.junit.Test;

import org.jmock.Expectations;
import org.jmock.integration.junit4.JUnitRuleMockery;

public class TurtleDriverTest {
    @Rule
    public JUnitRuleMockery context = new
        JUnitRuleMockery();

    private final Turtle turtle = context.mock(Turtle.
        class);

    // @Test methods here

}
```

jMock — Hello, world! (cont.)

```
@Test public void
goesAMinimumDistance() {
    final Turtle turtle2 =
        context.mock(Turtle.class, "turtle2");
    final TurtleDriver driver =
        new TurtleDriver(turtle1, turtle2); // set up

    context.checking(new Expectations() {{ // expectations
        ignoring (turtle2);
        allowing (turtle).flashLEDs();
        oneOf (turtle).turn(45);
        oneOf (turtle).forward(with(greaterThan(20)));
        atLeast(1).of (turtle).stop();
    }});

    driver.goNext(45); // call the code
    assertTrue("driver has moved",
               driver.hasMoved()); // further assertions
}
```

jMock — JUnit integration

```java
import static org.junit.Assert.*;
import org.junit.Rule;
import org.junit.Test;

import org.jmock.Expectations;
import org.jmock.integration.junit4.JUnitRuleMockery;

public class TurtleDriverTest {
    @Rule
    public JUnitRuleMockery context = new
        JUnitRuleMockery();
```

- `@Rule` is a JUnit annotation that subsumes `@Before`/`@After` annotations, by grouping together context managers for test methods
- `JUnitRuleMockery` is a JUnit rule for jMock↔JUnit integration
- you have to instantiate it to a (single) mockery object. `context` is a canonical—and reasonable, for readability—name for the instance variable

jMock — creating mock objects

```
private final Turtle turtle = context.mock(Turtle.class);
[...]
final Turtle turtle2 = context.mock(Turtle.class, "turtle2"
```

- we use the mockery to create mock objects
- like most mock frameworks, jMock heavily uses reflection to create mock objects...
- ...you have to pass an object representing the class you want to mock; you do so via the `.class` attribute
- jMock assigns mock object names and uses them in error messages
  - the canonical name is assigned to the first mock object of each type
  - you have to choose different names for extra objects
    - e.g., `"turtle2"` in the example above

jMock — expectation blocks

You group expectations together in expectation blocks:

```
context.checking(new Expectations() {{
    // expectations here, e.g.:
    oneOf (turtle).turn(45);
}});
```

What's this double brace syntax?

jMock — expectation blocks (cont.)

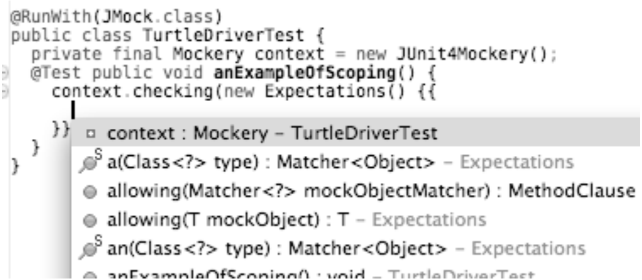
Unraveling jMock's syntax:

```
context.checking(
  new Expectations() {
    {
      // expectations here, e.g.:
      oneOf (turtle).turn(45);
    }
  }
);
```

- **new** Expectations new instance of...
- outer braces: anonymous subclass of `org.jmock.Expectations`
- inner braces: instance initialization block, which will be called after parent class constructor; within you can access:
  - `Expectation`'s instance methods
  - surrounding scope, with care (e.g., local variables must be **final**)

jMock — expectation blocks, discussion

Given that instance methods are visible by default in the initialization block, we can use them to build a Domain Specific Language (DSL) to define expectations, where we use method names as “words” of the language.



GOOS, Figure A.1

jMock — expectation DSL

Expectations have the following general form:

```
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));
```

Example

```
oneOf (turtle).turn(45); \\ turtle must be told once to turn 45
atLeast(1).of (turtle).stop(); \\ must be told 1+ to stop
allowing (turtle).flashLEDs(); \\ may be told 0+ times flash LED
allowing (turtle).queryPen(); will(returnValue(PEN_DOWN));
                \\ ditto + the turtle will always return PEN_DOWN
ignoring (turtle2);     \\ no expectations on mock object turtle2
```

note the peculiar use of spacing

jMock — invocation count

`exactly(n).of`	exactly n invocations
`oneOf`	= `exactly(1).of`
`atLeast(n).of`	$\geq n$ invocations
`atMost(n).of`	$\leq n$ invocations
`between(n,m).of`	$n \leq$ invocations $\leq m$

allowing	= `atLeast(0).of`, i.e., method can be invoked any number of time
ignoring	= `allowing`
never	= `atMost(0).of`, i.e., method must never be called, this is the default behavior for all mock object methods

allowing/ignoring/never can also be applied to entire objects, and composed together, e.g.:

```
allowing(turtle2);        // allow all method invocations...
never(turtle2).stop();  // ...except stop()
```

jMock — method invocation

The expected invocation counts—as well as other constraints, e.g., on method arguments—apply to method invocations. To specify the method you just "call" the method on the mock object.

```
oneOf (turtle).turn(45);          // matches turn() called with 45
oneOf (calculator).add(2, 2);    // matches add() called with 2 an
```

jMock — arguments constraints

Simple argument matching is done by simply providing the expected arguments, like in the previous example:

```
oneOf (turtle).turn(45);          // matches turn() called with 45
oneOf (calculator).add(2, 2);   // matches add() called with 2 an
```

In this case, arguments are compared for equality, i.e., using `.equals`.

For more flexible matching you can use the `with()` clause and argument matchers. . .

jMock — arguments matchers

`equal(`o`)`	o`.equals` arg
`same(`o`)`	o == arg
`any(Class<T>` t`)`	arg has[2] type t
`aNull(Class<T>` t`)`	ditto + arg is `null`
`aNonNull(Class<T>` t`)`	ditto + arg is *not* `null`

```
oneOf(calculator).add(with(equal(15)), with(any(int.class)));
// matches add() called with 15 and any other number
```

gotcha: either all arguments use `with()`, or none does

2. taking sub-typing into account

Hamcrest

Even more flexible matching—for jMock, JUnit, and more general use—is provided by the Hamcrest collection of matchers, e.g.:

- object
  - `hasToString` — test Object.toString
- numbers
  - `closeTo` — test floating point values are close to a given value
  - `greaterThan`, `greaterThanOrEqualTo`, `lessThan`, `lessThanOrEqualTo` — test ordering
- collections
  - `array` — test an array's elements against an array of matchers
  - `hasEntry`, `hasKey`, `hasValue` — test a map contains an entry, key or value
  - `hasItem`, `hasItems` — test a collection contains elements
  - `hasItemInArray` — test an array contains an element
- text
  - `equalToIgnoringCase` — test string equality ignoring case
  - `equalToIgnoringWhiteSpace` — test string equality ignoring differences in runs of whitespace
  - `containsString`, `endsWith`, `startsWith` — test string matching

`http://hamcrest.org/`, Free Software, BSD-licensed

jMock — actions

We are testing objects as peers conversing according to a protocol.

- we already have enough expressivity to express expectations on outgoing (sent) messages
- actions allow to express expectations on incoming (received) messages

You express actions within `will()` clauses placed after invocation counts. Some predefined actions are:

`returnValue(`v`)`	return value v
`throwException(`e`)`	throw exception e
`returnIterator(`c`)`	return iterator on collection c
`returnIterator(`$v_1, \ldots, v_n$`)`	return iterator on $v_1, \ldots, v_n$
`doAll(`$a_1, \ldots, a_n$`)`	perform all a_i actions

```
allowing (turtle).queryPen(); will(returnValue(PEN_DOWN));
    // queryPen can be invoked any number of times
    // at each invocation, it will return PEN_DOWN
```

jMock — sequences

Thus far, we can only express stateless protocols, where all expectations have the form "when you receive *foo*—no matter your state—do *bar*".

jMock offers two mechanisms to specify stateful protocols. The simplest are sequences.

- you can create multiple, independent sequences
- invocation counts can be assigned to sequences
- invocations in the same sequence must occur in order
  - specifically: *all* invocations must occur before next method

```
final Sequence drawing = context.sequence("drawing");

allowing (turtle).queryColor(); will(returnValue(BLACK));
atLeast(1).of (turtle).forward(10);    inSequence(drawing);
oneOf (turtle).turn(45);               inSequence(drawing);
oneOf (turtle).forward(10);            inSequence(drawing);
```

jMock — state machines

A more general mechanism to specify stateful protocols are state machines (think of sequences as strictly linear state machines). You can create multiple, independent state machines. You set/query the current state using postfix clauses:

`when(stateMachine.is("state"))`	invocation must occur within *state*
`when(stateMachine.isNot("state"))`	ditto, negated
`then(stateMachine.is("state"))`	change to *state*

```java
final States pen = context.states("pen").startsAs("up");

allowing (turtle).queryColor(); will(returnValue(BLACK));
allowing (turtle).penDown();        then(pen.is("down"));
allowing (turtle).penUp();          then(pen.is("up"));
atLeast(1).of (turtle).forward(15); when(pen.is("down"));
one (turtle).turn(90);              when(pen.is("down"));
one (turtle).forward(10);           when(pen.is("down"));
```

Outline

Outline

Too many expectations

Too many expectations will make very difficult to understand what is under test, as opposed to setup code, e.g.:

```
@Test public void
decidesCasesWhenFirstPartyIsReady() {
    context.checking(new Expectations(){{
        oneOf(firstPart).isReady(); will(returnValue(true))
        oneOf(organizer).getAdjudicator();
          will(returnValue(adjudicator));
        oneOf(adjudicator).findCase(firstParty, issue);
          will(returnValue(case));
        oneOf(thirdParty).proceedWith(case);
    }});
    claimsProcessor.adjudicateIfReady(thirdParty, issue);
}
```

i.e., everything looks equally important

Too many expectations (cont.)

Tips to improve:

- spot errors in the specification: do all methods need to be called exactly once to be correct? (e.g., query methods can be safely called multiple times)
- distinguish between: stubs, simulations of real behavior, expectations, and assertions

$$\Downarrow$$

```
@Test public void
decidesCasesWhenFirstPartyIsReady() {
    context.checking(new Expectations(){{
        allowing(firstPart).isReady(); will(returnValue(tru
        allowing(organizer).getAdjudicator();
          will(returnValue(adjudicator));
        allowing(adjudicator).findCase(firstParty, issue);
          will(returnValue(case));
        oneOf(thirdParty).proceedWith(case);
    }});
    claimsProcessor.adjudicateIfReady(thirdParty, issue);
}
```