

# Méthodes de Test

## Cours 3 — Mock Objects

Stefano Zacchioli  
zack@pps.univ-paris-diderot.fr

Laboratoire PPS, Université Paris Diderot

2013-2014

URL <http://upsilon.cc/zack/teaching/1314/methctest/>  
Copyright © 2013 Stefano Zacchioli  
License Creative Commons Attribution-ShareAlike 3.0 Unported License  
<http://creativecommons.org/licenses/by-sa/3.0/>



- 1 Testing object-oriented systems
- 2 Object mocking
- 3 Maintaining TDD — part 2
  - Test smells
  - Test readability
  - Test diagnostics

## 1 Testing object-oriented systems

## 2 Object mocking

## 3 Maintaining TDD — part 2

- Test smells
- Test readability
- Test diagnostics

# 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

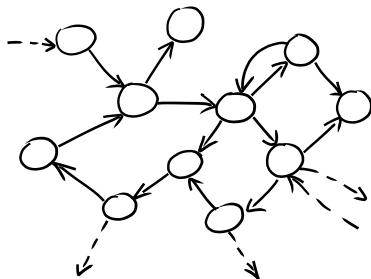
- invoke method *m* on object *obj* ~ send message *m* to object *obj*

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 has **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 entity**, 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 object dependencies (see cours-2)
- 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.

## Tell, don't ask — redux

“Train wreck” code: series of getters chained together like the carriages in a train:

```
((EditSaveCustomizer) master.getModelisable()
    .getDockablePanel()
    .getCustomizer()
    .getSaveItem().setEnabled(Boolean.FALSE.booleanValue()));
```

what it actually means:

```
master.allowSavingOfCustomisations();
```

### “Tell, don't ask” principle

**Don't ask** (recursively) access to objects internals that allow you to perform a specific operation. Rather, **tell** the (outermost) object to do something on your behalf; let it do the same, recursively, as needed.

This makes your tests (and code in general) **more resistant to changes in object organization**.



## Tell, don't ask — redux (cont.)

When you really have to ask, do so explicitly via **well-defined query methods** (i.e. queries that have **clear names** and **well-defined semantics**):

```
public void reserveSeats(ReservationRequest request) {
    for (Carriage carriage : carriages) {
        if (carriage.getSeats().getPercentReserved() < percentReservedBa
            request.reserveSeatsIn(carriage);
        return;
    }
    request.cannotFindSeats(); }
```



```
public void reserveSeats(ReservationRequest request) {
    for (Carriage carriage : carriages) {
        if (carriage.hasSeatsAvailableWithin(percentReservedBarrier)) {
            request.reserveSeatsIn(carriage);
            return;
        }
    }
    request.cannotFindSeats(); }
```



## Example — testing observer

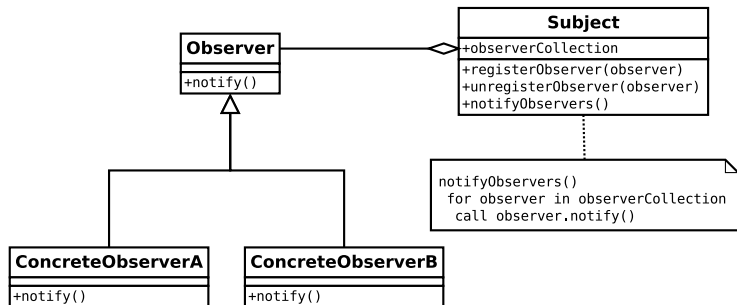
### Observer design pattern — reminder

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](https://en.wikipedia.org/wiki/Observer_pattern)

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

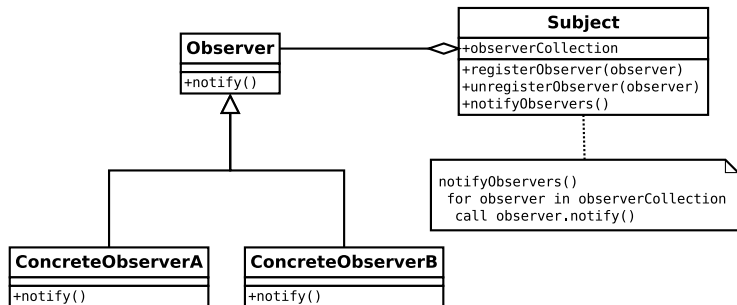
# Observer — what to test



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

what should we test?

# Observer — what to test

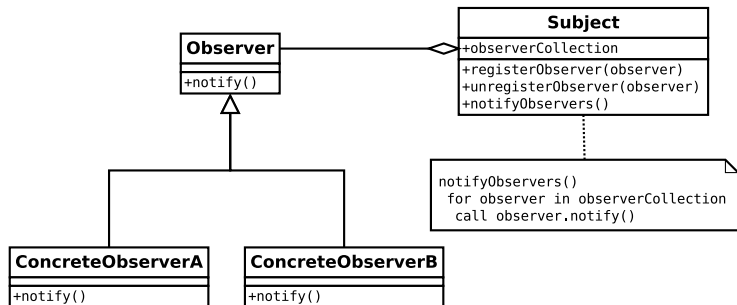


<https://en.wikipedia.org/wiki/File:Observer.svg>

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 void~~ (?!?!)
- ~~register/unregister round trip~~ (how w/o internal state?)
- ...

## Observer — what to test (cont.)

adding observers “works”, i.e.:

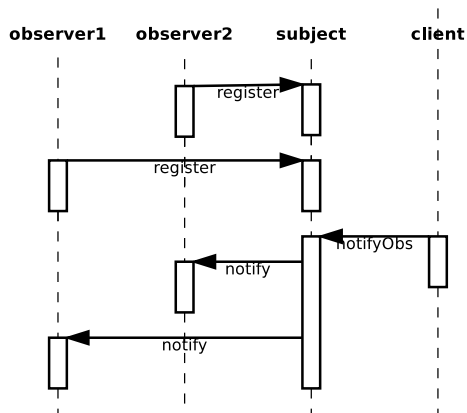


Figure : 1 subject with 2 observers, sequence diagram

## 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**

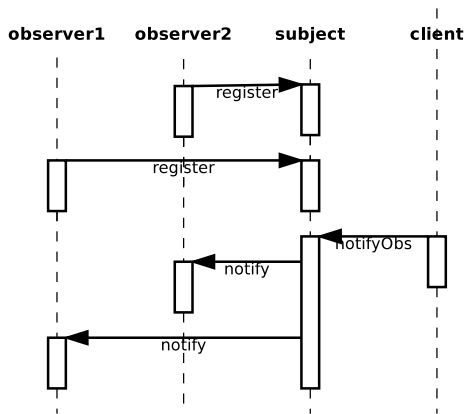


Figure : 1 subject with 2 observers, sequence diagram



## 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**

let's try testing this with JUnit...

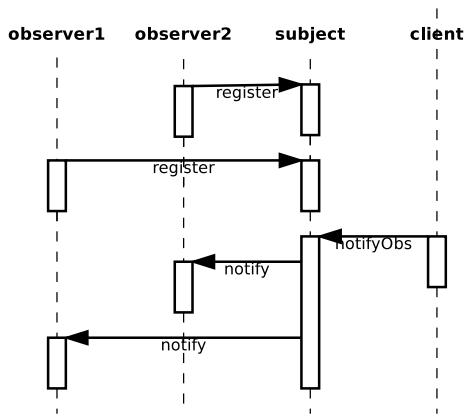


Figure : 1 subject with 2 observers, sequence diagram

## Example — interfaces

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

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

## Example — implementation

```
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
    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.)

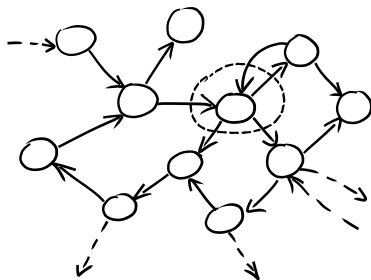
### Discussion:

- 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
- 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
  - ▶ . . .



GOOS, Figure 2.4



- 1 Testing object-oriented systems
- 2 Object mocking**
- 3 Maintaining TDD — part 2
  - Test smells
  - Test readability
  - Test diagnostics

# 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 wrt mock objects, i.e. which **messages** the tested object exchanges with mock objects.

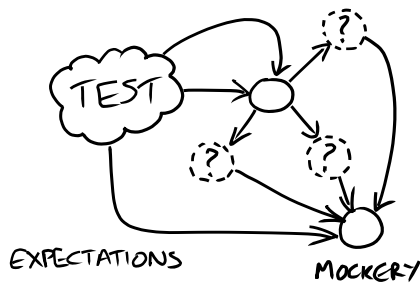


GOOS, Figure 2.5

# 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 real objects, including 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<sup>1</sup>

```
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
}
```

---

<sup>1</sup>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 wrt previous test (highlighted in the code):

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

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://jmock.org/>
- jMock is Free Software, BSD-licensed

## jMock — Hello, world!

```
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

```
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);  
        }  
    }  
);
```

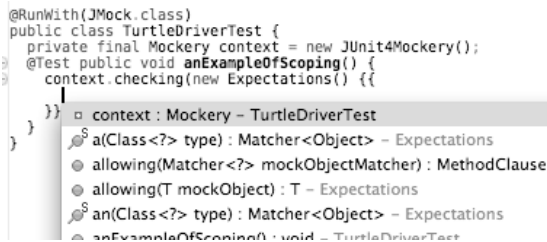
- **new** Expectations new instance of...
- outer braces: **anonymous subclass** of org.jmock.Expectations
- inner braces: **instance initialization block**, which will be called after parent class constructor; within you can access:
  - ▶ Expectation's instance methods
  - ▶ surrounding scope, with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

## jMock — expectation blocks, discussion

Expectation blocks use a pretty clever hack on Java scoping rules.

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.

```
@RunWith(JMock.class)
public class TurtleDriverTest {
    private final Mockery context = new JUnit4Mockery();
    @Test public void anExampleOfScoping() {
        context.checking(new Expectations() {{
            //
        }})
    }
}
```



GOOS, Figure A.1

**Figure :** as a result, code-completion in IDE can be very precise

## 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 LEDs  
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`  $n \leq \text{invocations} \leq m$

`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 and 2
```



## 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 and 2
```

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 has2 type t`  
`aNull(Class<T> t)`    `ditto + arg is null`  
`aNotNull(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

---

<sup>2</sup>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:

<code>returnValue(v)</code>	return value $v$
<code>throwException(e)</code>	throw exception $e$
<code>returnIterator(c)</code>	return iterator on collection $c$
<code>returnIterator(v<sub>1</sub>, ..., v<sub>n</sub>)</code>	return iterator on $v_1, \dots, v_n$
<code>doAll(a<sub>1</sub>, ..., a<sub>n</sub>)</code>	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:

<code>when(stateMachine.is("state"))</code>	invocation must occur within <i>state</i>
<code>when(stateMachine.isNot("state"))</code>	ditto, negated
<code>then(stateMachine.is("state"))</code>	change to <i>state</i>

```
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

- 1 Testing object-oriented systems
- 2 Object mocking
- 3 Maintaining TDD — part 2
  - Test smells
  - Test readability
  - Test diagnostics

# Outline

- 1 Testing object-oriented systems
- 2 Object mocking
- 3 Maintaining TDD — part 2**
  - Test smells
  - Test readability
  - Test diagnostics



## Mocking concrete classes

Mantra: **mock interfaces, not concrete classes.**

- **jMock enforces this**; to mock concrete classes you should use the `ClassImposteriser` extension

Here is an example of concrete class mocking (by hand):

```
public class MusicCentreTest {  
    @Test public void  
    startsCdPlayerAtTimeRequested() {  
        final MutableTime scheduledTime = new MutableTime();  
        CdPlayer player = new CdPlayer() {  
            @Override  
            public void scheduleToStartAt(Time startTime) {  
                scheduledTime.set(startTime);  
            }  
        }  
        MusicCentre centre = new MusicCentre(player);  
        centre.startMediaAt(LATER);  
        assertEquals(LATER, scheduledTime.get());  
    }  
}
```

## Mocking concrete classes (cont.)

Problems with mocking concrete classes:

- 1 the **relationship** between CdPlayer and MusicCentre **remains implicit**
  - ▶ whereas if you have an interface, you re be forced to *think* about that relationship and define it
  - ▶ in this example, it is **too much** for a MusicCentre to depend on a CdPlayer, as it only needs the start and stop methods
  - ▶ wouldn't it be better to define a ScheduledDevice interface, and have MusicCentre depend on it? (yes)
- 2 being forced to think about interfaces also forces you to **name interfaces**. That, in turn, helps in clarifying the domain

## Mocking concrete classes (cont.)

Problems with mocking concrete classes:

- 1 the **relationship** between CdPlayer and MusicCentre **remains implicit**
  - ▶ whereas if you have an interface, you re be forced to *think* about that relationship and define it
  - ▶ in this example, it is **too much** for a MusicCentre to depend on a CdPlayer, as it only needs the start and stop methods
  - ▶ wouldn't it be better to define a ScheduledDevice interface, and have MusicCentre depend on it? (yes)
- 2 being forced to think about interfaces also forces you to **name interfaces**. That, in turn, helps in clarifying the domain

In some cases you're **forced to mock concrete classes**

- e.g. when testing code you can't change
- best practices for those (rare!) cases:
  - 1 only override public methods, not internal features
  - 2 do not make public internal features, otherwise you tie the test to the current implementation

# Don't mock values

Mantra: **don't mock values**, just create instances and use them.

```
@Test public void sumsTotalRunningTime() {
    Show show = new Show();
    Video video1 = context.mock(Video.class); // don't do this
    Video video2 = context.mock(Video.class);
    context.checking(new Expectations(){{
        one(video1).time(); will(returnValue(40));
        one(video2).time(); will(returnValue(23)); }});
    show.add(video1);
    show.add(video2);
    assertEquals(63, show.runningTime())
}
```

To identify a value (heuristics):

- it's **immutable**
- you can't think of a **meaningful name** for the corresponding interface (names like `VideoImpl` don't count)

Note: if a value is too hard to build, don't mock it, **write a builder**

## 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



@Test **public void**

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

# Outline

- 1 Testing object-oriented systems
- 2 Object mocking
- 3 Maintaining TDD — part 2**
  - Test smells
  - Test readability**
  - Test diagnostics

## What not how

Mantra: move **out of the test** everything that doesn't contribute to the **description of the feature under test**.

- use **helpers** methods, **queries**, **builders**
- write **custom matchers** to encode **predicates** for assertions and expectations

```
assertThat(instruments ,  
           hasItem(instrumentWithPrice(greaterThan(81))));
```

```
private Matcher<? super Instrument>  
instrumentWithPrice(Matcher<? super Integer> priceMatcher) {  
    return new FeatureMatcher<Instrument, Integer>(  
        priceMatcher, "instrument at price", "price") {  
        @Override  
        protected Integer featureValueOf(Instrument actual) {  
            return actual.getStrikePrice();  
        }  
    };  
}
```



# Assertions and expectations

Mantra: **don't assert/expect too much.**

Only assert/expect at a **level of detail** corresponding to the correct behavior of the code under test.

E.g.: is it really relevant that a code raise a *specific* exception or would any exception do?

It depends. Ask yourself the question!

In the latter case prefer code like:

```
oneOf(failureReporter).cannotTranslateMessage(  
    with(SNIPER_ID), with(badMessage),  
    with(any(RuntimeException.class)));
```

# Outline

- 1 Testing object-oriented systems
- 2 Object mocking
- 3 Maintaining TDD — part 2**
  - Test smells
  - Test readability
  - Test diagnostics**

## Highlight detail with matches

When using custom matchers (Hamcrest or others), you often have the possibility to **customize the mismatch messages**. That is a very powerful tool to have **effective diagnostic** in case of failure (“listen to the test,” remember?), e.g.:

```
Expected: a collection containing instrument at price a value  
         greater than <81>  
but: price was <50>, price was <72>, price was <31>
```

See API details at, e.g. [https://code.google.com/p/hamcrest/wiki/Tutorial#Writing\\_custom\\_matchers](https://code.google.com/p/hamcrest/wiki/Tutorial#Writing_custom_matchers)

## Enforcing expectations

If your test has **both assertions and expectations**, the **order** in which they are verified matters.

- in most mock frameworks, **expectations are checked at the end** of the test body
- if a message exchange didn't work properly and returned a wrong value, it might trigger **bogus assertion failures**
- in jMock, you can **anticipate expectation verification** using `assertIsSatisfied` on the mockery

```
context.assertIsSatisfied();  
assertThat(result, equalTo(expectedResult));
```

remember: always “watch the test fail” before making it pass, it will point you at the need of using `assertIsSatisfied`

# Tracer object

Some times you just need to **follow an object through the protocol**. You can do so by using **tracer objects**, which have no other purpose, nor content, than being traced. E.g.:

```
final Lineltem item1 = context.mock(Lineltem.class, "item1");  
final Lineltem item2 = context.mock(Lineltem.class, "item2");  
final Billing billing = context.mock(Billing.class);  
@Test public void requestsInvoiceForPurchasedItems () {  
    context.checking(new Expectations () {{  
        oneOf(billing).add(item1);  
        oneOf(billing).add(item2); }});  
    customer.purchase(item1, item2);  
    customer.requestInvoice(billing);  
}
```

Possibly resulting **diagnostic**:

```
not all expectations were satisfied. Expectations:  
  expected once, already invoked 1 time: billing.add(<item1>)  
  ! expected once, never invoked: billing.add(<item2>>)  
what happened before this: billing.add(<item1>)
```