

Software Craftsmanship 2009

Nat Pryce - nat.pryce@gmail.com

Version: 1.1
notes

Course code: software-craftsmanship-2009

Software Craftsmanship 2009

Copyright © 2008 - 2009 Nat Pryce

Course materials may not be copied or distributed in any form without the prior written permission of the copyright holders.

Contents:

Software Craftsmanship 2009.....4

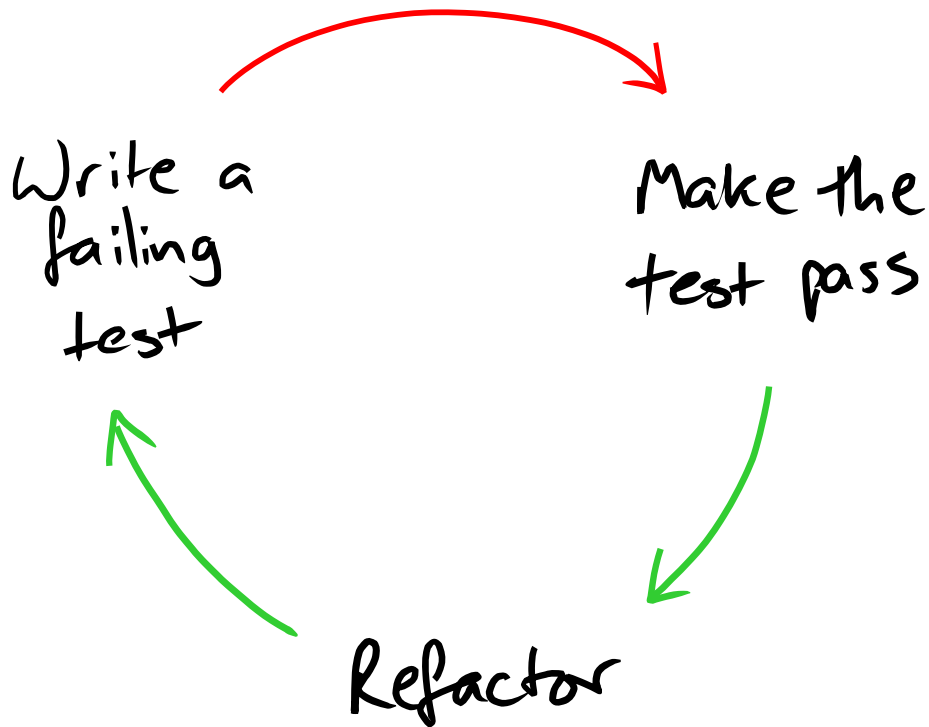
Unit 1

Software Craftsmanship 2009

1. Test-Driven Development of Asynchronous Systems

Test-Driven Development of Asynchronous Systems

Basic TDD Cycle



Notes:

TDD's golden rule is: *never write new functionality unless you have a failing test.*

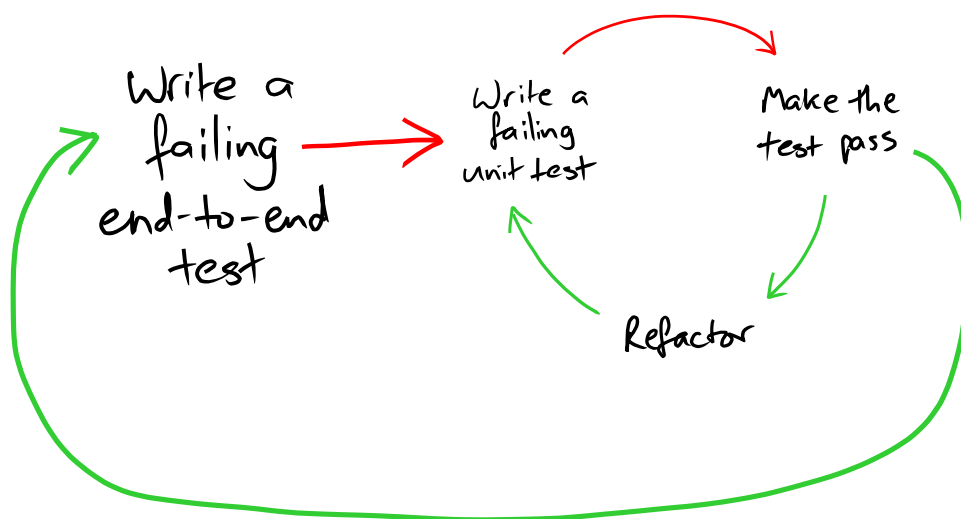
However, this is a very simplistic description of TDD. How do you start the cycle?

Typically, people leap right in, writing unit tests for classes deep in the internals of their system: domain model classes, for example. This leads to integration problems later.

Integration:

- Do the modules of the system integrate with one another?
- Does the system integrate into its operating environment?

Start with End-to-End Tests



Notes:

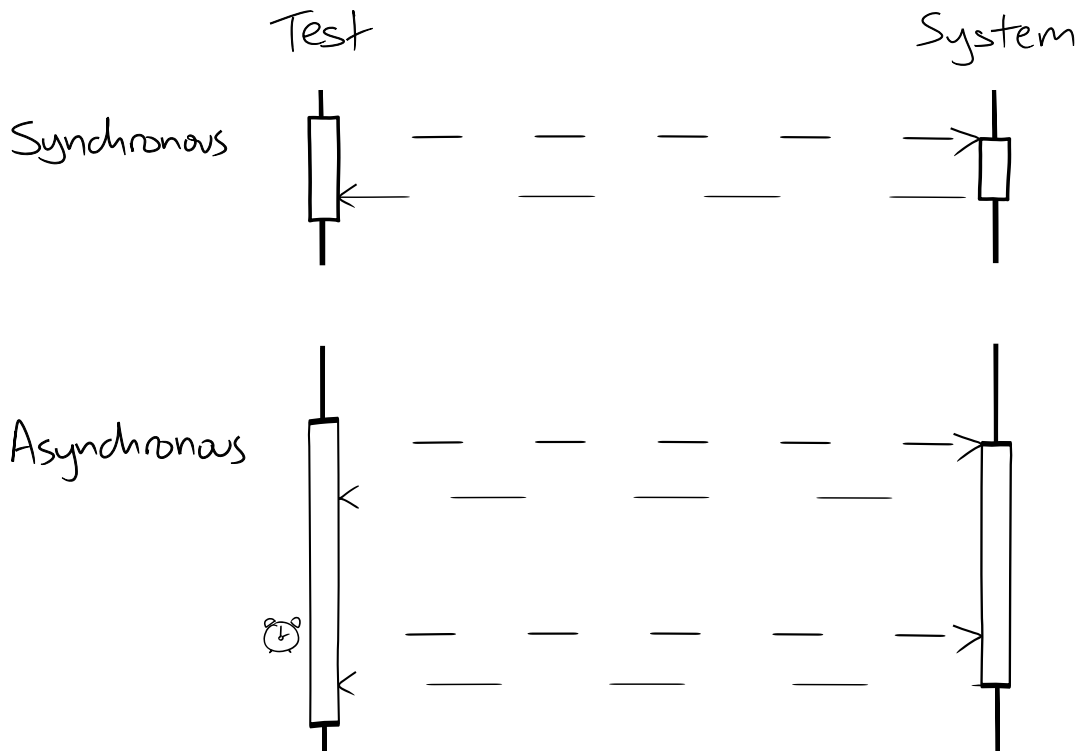
Address integration problems *first*.

Kick off the whole process by writing a failing test that demonstrates that the *system* doesn't do what is required.

You can then use the core TDD cycle to write the code that makes the system pass the end-to-end test.

"End-to-end" refers both to the system and the process of development, packaging and deployment. The end-to-end test run is triggered by developers checking in code. It then checks out the code, builds deployable packages, deploys the system into a production-like environment and tests the system by driving its "edges": the user interface, feeds to and from third party systems, reporting and monitoring interfaces, etc.

Synchronous vs. Asynchronous Tests



Notes:

When writing system tests or integration tests for concurrent code, you have to cope with a system that executes asynchronously with respect to the test.

Synchronous test: control returns to test after system under test completes. Errors are detected immediately.

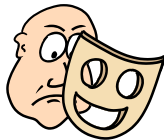
Asynchronous test: control returns to test while system under test continues. Errors are swallowed by the system under test.

Failure is detected by the system not entering an expected state or sending an expected notification within some timeout.

Common Problems



Flickering Tests



False Positives



Slow Tests



Messy Tests

Notes:

An asynchronous test must synchronise with the system under test, or problems occur:

Flickering Tests: tests usually pass but fail occasionally, for no discernible reason, at random, usually embarrassing, times. As the test suite gets larger, more runs contain test failures. Eventually it becomes almost impossible to get a successful test run.

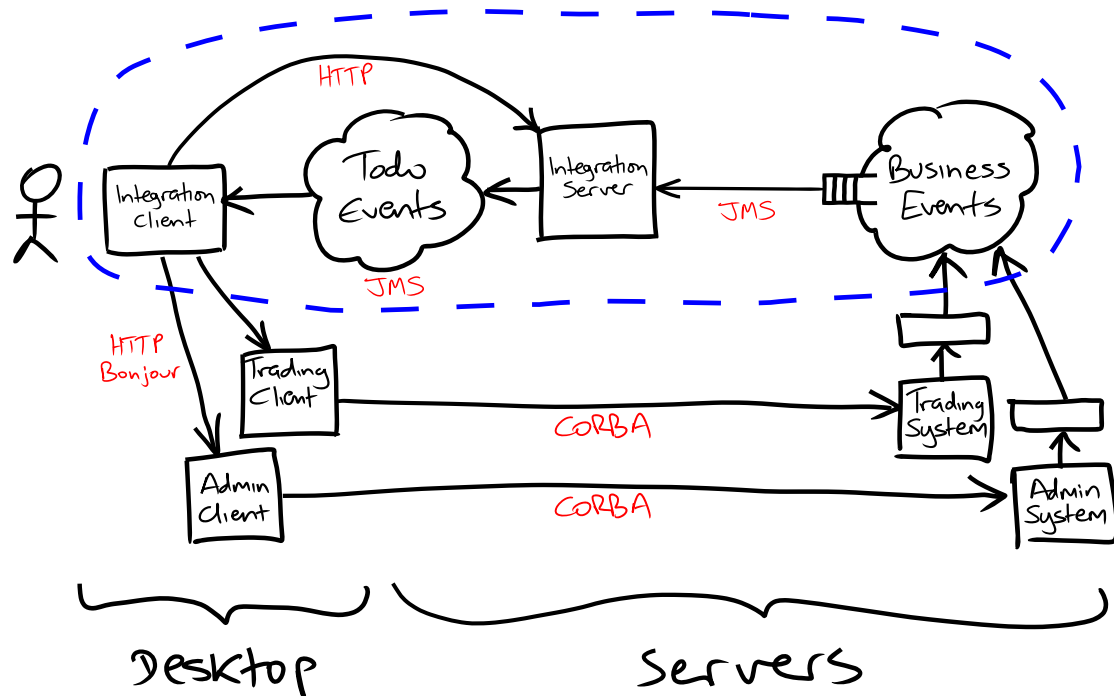
False Positives: the tests pass, but the system doesn't really work.

Slow Tests: the tests are full of sleep statements to let the system catch up. One or two subsecond sleeps in one test is not noticed, but when you have thousands of tests every second adds up to hours of delay.

Messy Tests: scattering ad-hoc sleeps and timeouts throughout the tests makes it difficult to understand the test: what the test is testing is obscured by how it is testing it.

We'll now look at how we managed to avoid these problems when using TDD to build three different systems.

System: Loan-o-matic



Notes:

An application for an investment bank that integrates two legacy systems: one for administering corporate loans, another for trading loans in the secondary market. Financial regulations mandate that loan business must be conducted by fax. Therefore the process is mostly manual. To integrate these two legacy systems, our system had to notify administrators when work was outstanding: to pay out money, chase up payments. Basically, an automatic to-do list.

Our "to-do" system deploys "agents" that monitor the two systems. When an agent detects a significant change in the system it publishes a "business event". The server interprets business events by creating, modifying or deleting to-do items.

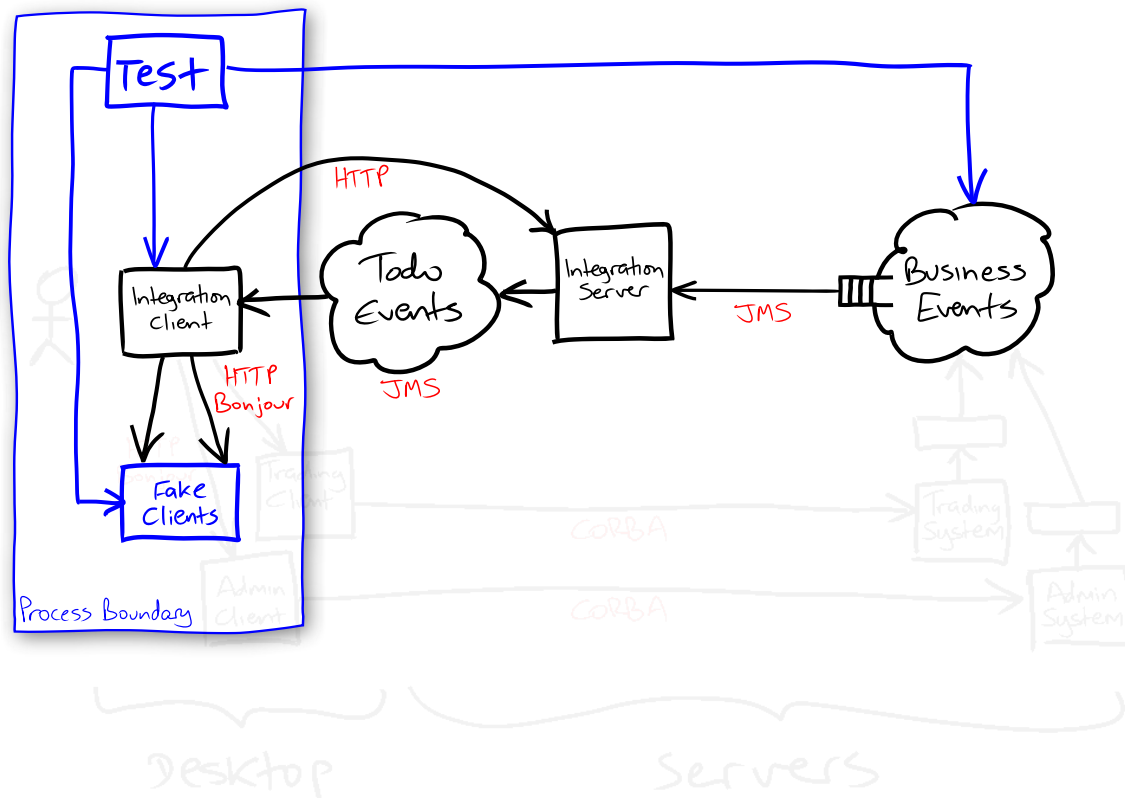
To-do items are broadcast out to clients on the users' desktops. The user can action a to-do item in the client, which opens a screen in the client of the appropriate legacy system and fills in whatever fields it can. When the user submits the screen, the legacy system changes, the agents detect the change, the server updates the to-do items, transmits the change to the clients, etc.

What caused us difficulties?

- **GUI Testing:** our end-to-end tests exercised the GUI. The tests had to cope with Swing's multithreaded, event-driven design. Don't know when Swing has finished processing an event.
- **Distributed Event Processing:** system behaviour triggered by business events but observed at the GUI. Don't know how long it will take for the system to receive and process an event.

- **Legacy:** (off topic) the blue boundary shows what we could write using TDD, starting with end-to-end tests. The parts that interfaced with the legacy systems could only be tested using unit tests or manual testing.

Testing Loan-o-matic



Notes:

System tests drove the system through the GUI and by sending "business events" by JMS. Also faked out the other clients on the desktop to test that the "remote control" interface was called as expected.

Client ran in the same address space as the tests. Made it easier to inspect the state of the GUI. The GUI was controlled by sending native input events with the AWT robot. This made it trivial to use custom GUI components.

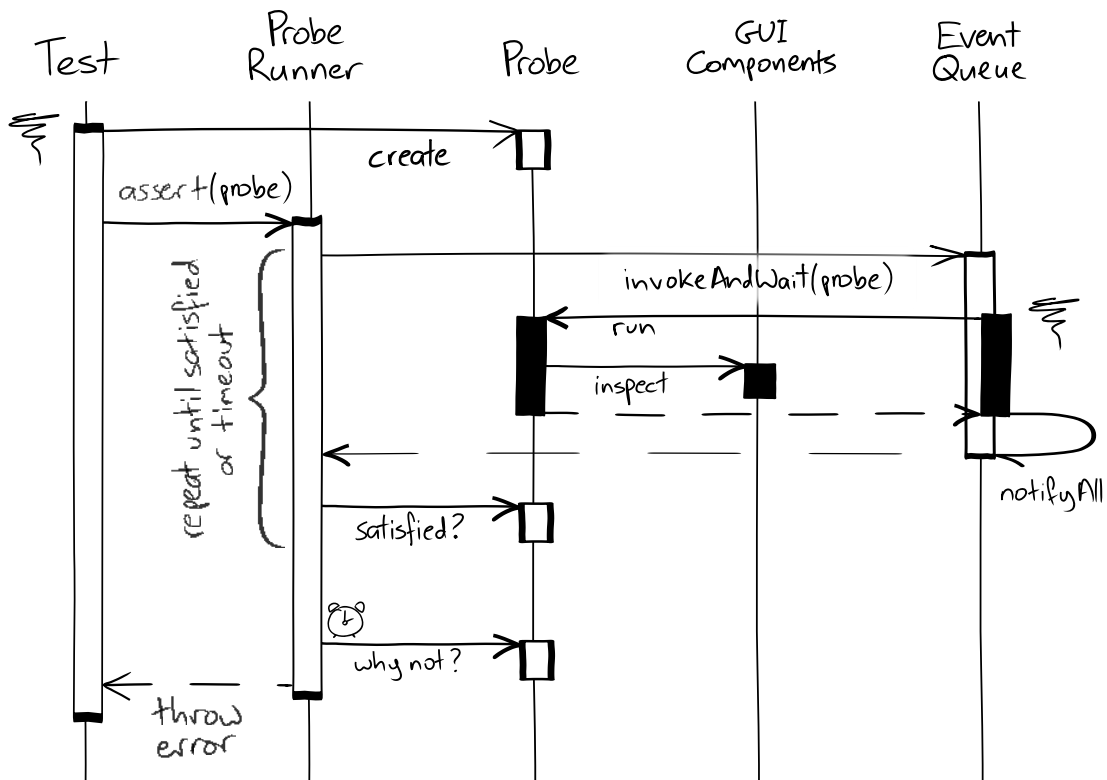
To cope with asynchrony, we wrapped the GUI in "drivers" that only let the test perform input actions or assert the state of the GUI. Behind the scenes, the drivers synchronised with the Swing event dispatch threads, polled the state of the GUI components, and failed the test if the asserted state did not occur within a timeout.

JUnit's assertions were not flexible enough. To test against the GUI, we had to pass assertions from the test thread to the GUI's event dispatch thread. An assertion had to evaluate itself on the dispatch thread but report failures by throwing exceptions in the test thread.

This was implemented as "probes" (objects that could be passed between threads) and "probe runners" (objects that knew how to pass a probe to the other thread and wait for it to complete, perform polling and timeouts, and report test failures).

Because our tests had to cope with Swing's concurrent, event-driven architecture, we got tests that coped with our system's concurrent, event-driven architecture for free. And vice versa.

Probes and Probe Runners



Notes:

A Probe is responsible for inspecting the GUI components to capture a snapshot of their state. It can report whether that snapshot satisfies some acceptance criteria and describe why it does not (if that is the case).

A ProbeRunner is responsible asserting that a Probe is satisfied within an acceptable timeout. It passes the Probe from the test thread to the GUI thread so that the probe can safely inspect the GUI components, and passes it back to the test thread again so that results can be reported on the test thread. The ProbeRunner repeatedly performs the probe until it is satisfied (in which case the assertion succeeds) or the timeout is met (in which case the assertion fails and is reported as an exception on the test thread).

This is implemented as an open source library called Window Licker.

Example Test of Loan-o-matic

```
@Test
public void aTradeClosedByAUSUserDoesNotAppearToAUKUser()
    throws Exception
{
    when(aTradeIsSettledInStealth().with(
        anAllocation().withAUTradeMarketAssociation()));

    client.currentTodoList().hasRowCount(1);

    client.currentTodoList().selectRow(0);
    client.performTodoItemAction(REQUEST_CLOSE);

    client.currentTodoList().hasRowCount(0);

    client.stop();

    startTheClientAndLogInAs("UK");

    client.show("Settled trades");
    client.currentTodoList().hasRowCount(0);
}
```

Notes:

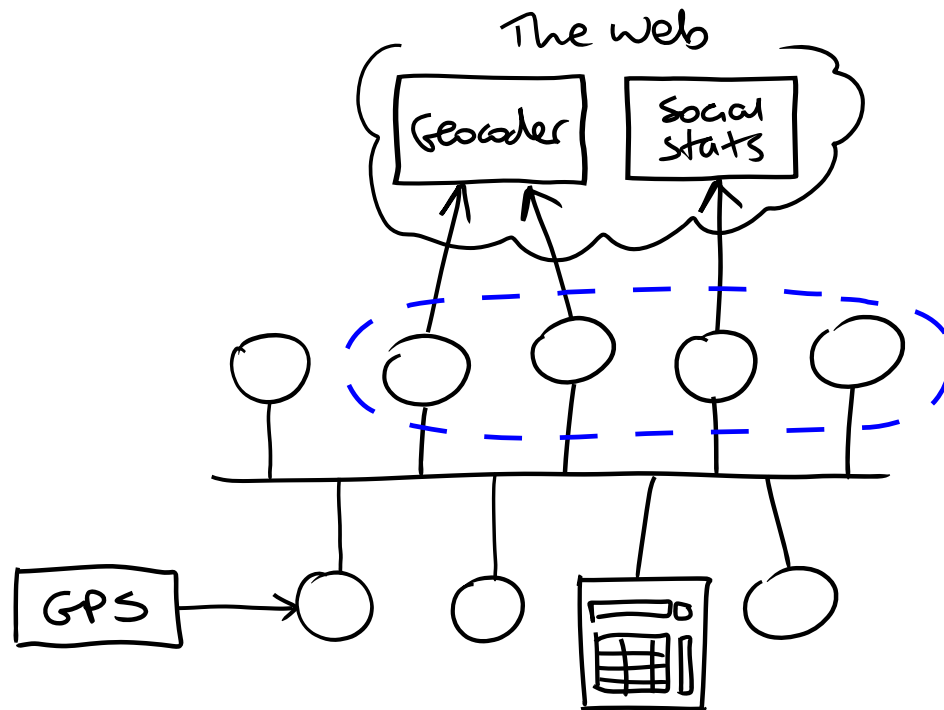
Moved all synchronisation out of the test classes, into reusable abstractions. Base test class defined common instances required for all tests and syntactic sugar.

The *client* variable is a driver that controls the GUI. It creates drivers that automate tasks that the user can perform with the GUI. (This shields the tests from changes to the GUI structure).

The *when* method sends a business event onto the input topic.

This API evolved through continual refactoring. We started by writing everything in the JUnit tests and then factored commonality out into new classes, renamed methods to express what is being tested, etc. We built the testing mechanisms and API alongside the system as the system grew.

System: React-o-matic



Notes:

A system for context-aware computing.

A service-oriented architecture in which services communicate by content-based pub-sub messaging.

Applications are implemented as a set of components that are plugged into a distributed message bus. Each process runs multiple components and the message bus of each process is federated with a device-wide bus, and that bus can be federated with buses on other devices.

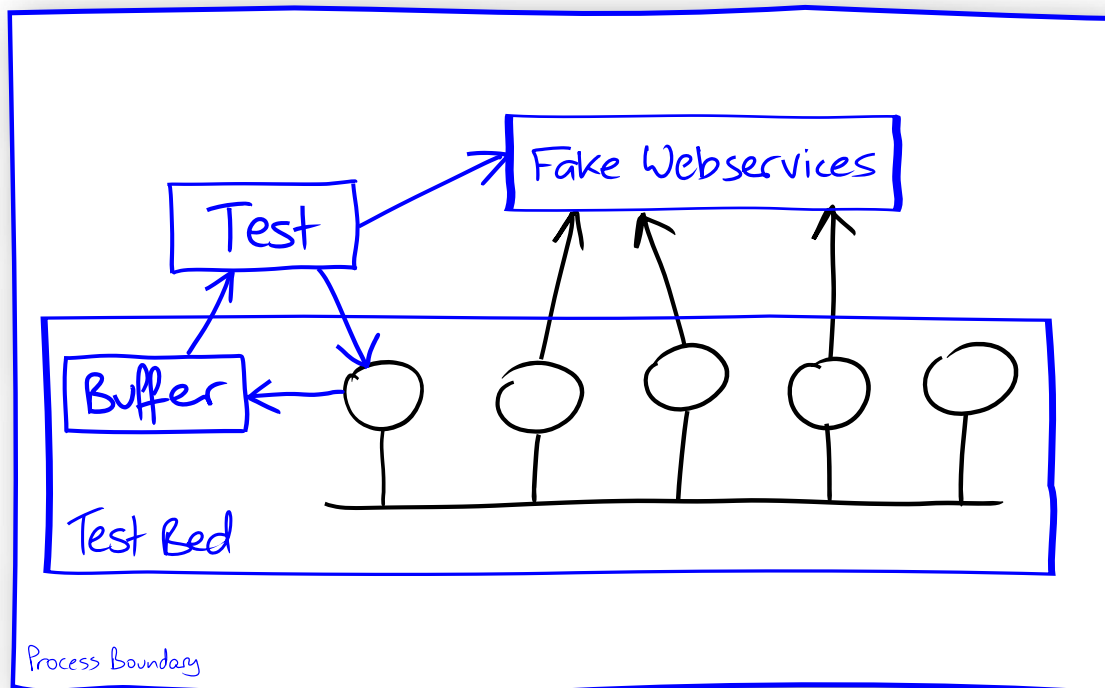
Components publish data read from hardware devices, translate primitive events into events with richer semantics, and consume events to act upon, display or record information.

For example, a service might report weather forecasts at the user's current location by using a geocoding service to translate position events (from a position tracking service) into current location events containing the post-code, and translate post-code events to weather reports obtained from a webservice.

What caused us difficulties?

- **Concurrent Event Processing:** events are dispatched to one or more subscribers that process the event concurrently before publishing new events.

Testing React-o-matic



Notes:

Testing a service is done with a "test bed", which is instantiated by a JUnit test.

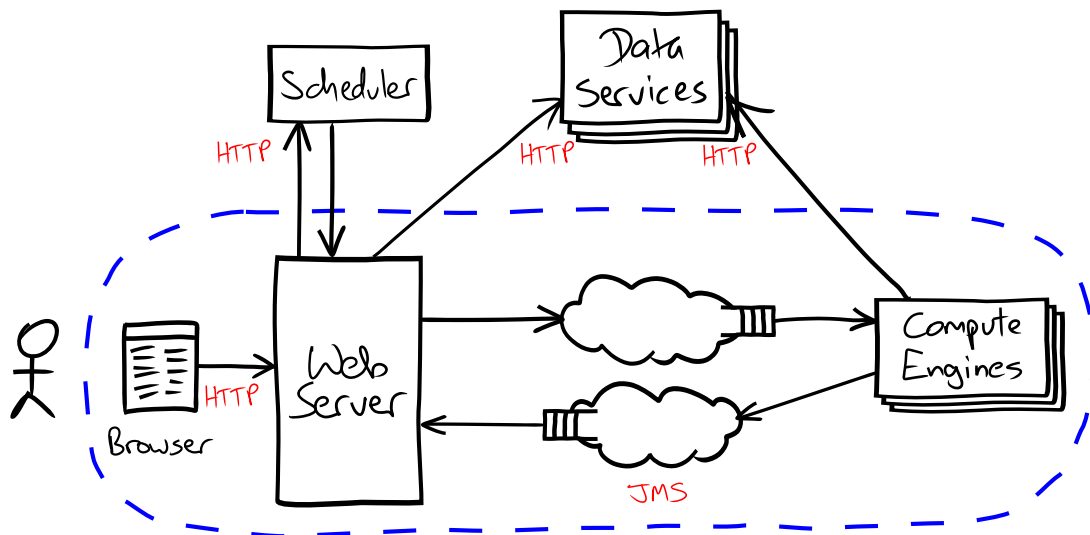
A test bed contains an in-memory event bus for speed. The event bus guarantees the same ordering and reliability semantics for local and remote message delivery.

The test-bed provides an API through which the test can send messages onto the bus to drive test scenarios and captures a trace of all other messages sent over the bus in an in-memory buffer.

The test creates service components and connects them to the bus, sends them messages and asserts that expected responses appear in the trace.

Assertions are implemented by waiting for the trace to have the expected contents. If an expected response is not in the trace, the test thread waits (with a timeout). Every time a message is delivered to the buffer, the deliver thread signals the test thread, which rechecks the contents of the trace. If the test thread times out, it fails the test.

System: Risk-o-matic



Notes:

System for performing complex financial calculations on a grid.

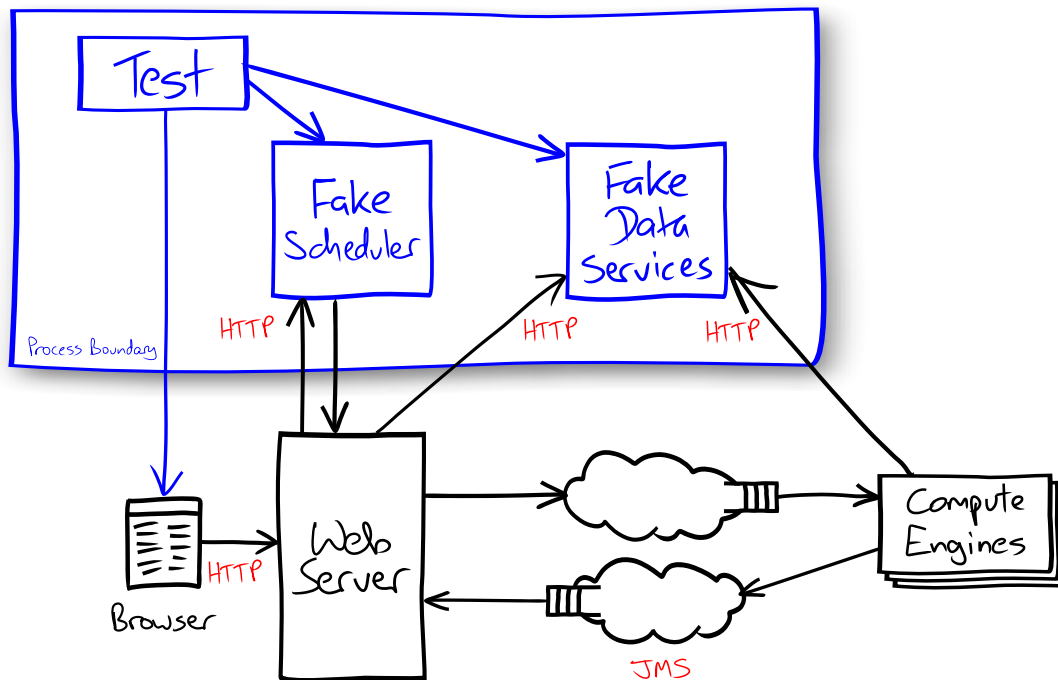
AJAX front-end written with GWT. Lets users define and schedule calculation jobs or execute jobs immediately.

Jobs to be executed are submitted to the execution engine by sending a message on a queue. The engine publishes messages on a topic to report progress of running jobs.

Progress is displayed in the AJAX GUI. When the job has finished, the final report can be downloaded from a link in the GUI, or the report is delivered to a final recipient by mail, messaging, etc.

An example test might be that when the user schedules a regular calculation job, the system executes that job as scheduled and the user can download the report from the web site after the job has completed.

Testing Risk-o-matic



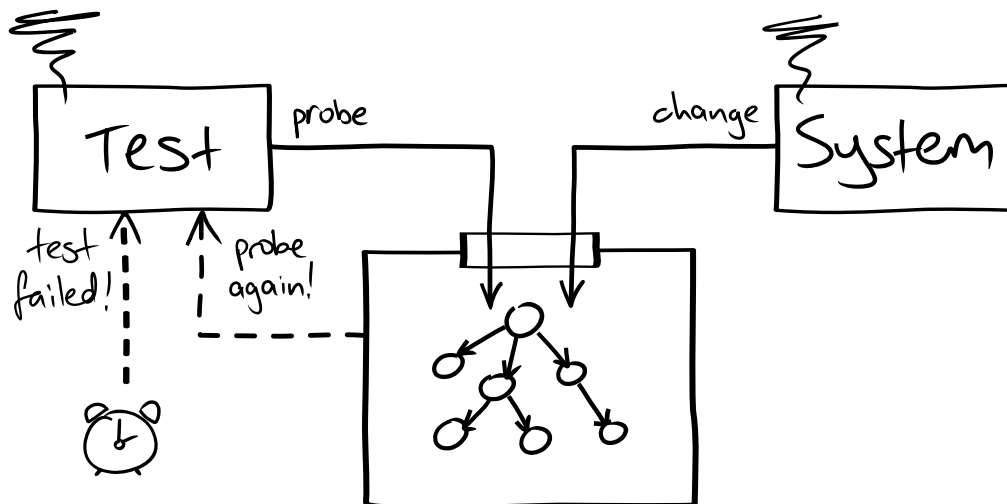
Notes:

Similar to Loan-o-matic, except that the GUI runs in a web browser, not a Java rich client. So, the test examines a browser's DOM tree, not a tree of Swing components.

All the web testing APIs we looked at assume a synchronous model and do not cope well with asynchronous Javascript. If they do support it, they do so with ad-hoc synchronisation (messy, unreliable) or by hooking in to the browser's behaviour, which does not cope with asynchrony elsewhere in the system.

We were able to combine WebDriver and Windowlicker into a test framework that made testing AJAX applications very easy.

Common Test Structure



Notes:

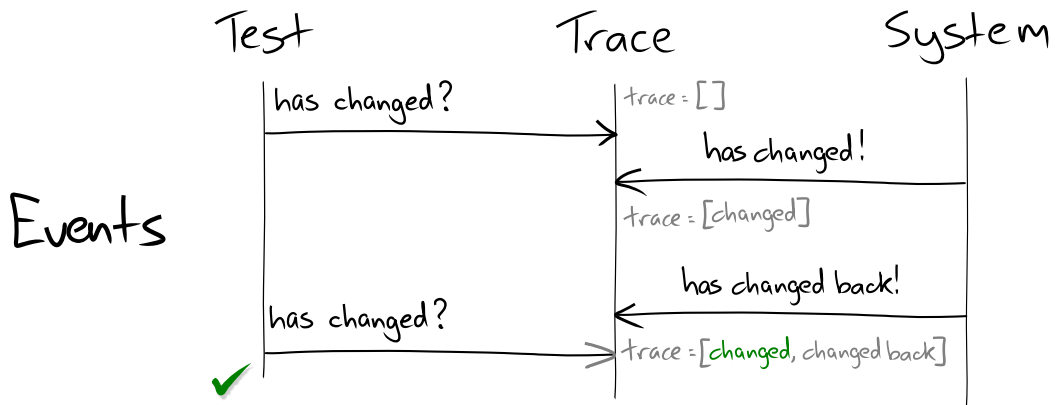
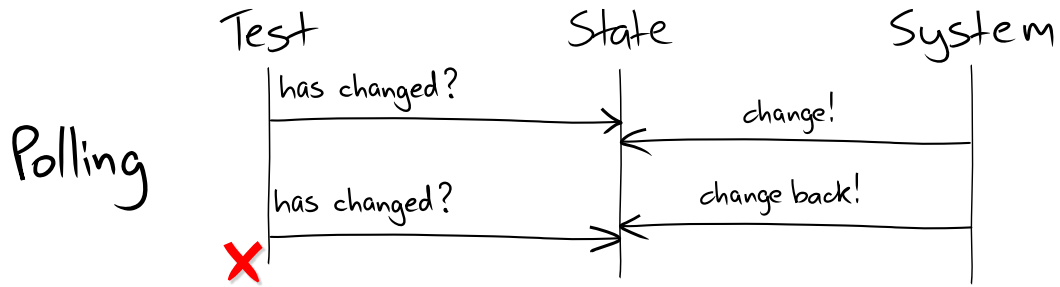
Common test structure:

- **Shared, Thread-Safe State:** updated by the system, examined by the test.
- **Look for Success, Not Failure:** wait for state to enter expected state, indicating successful execution. (Contrast with synchronous tests, which look for invalid states).
- **Succeed Fast:** detect success as fast as possible by re-examining the shared state on either:
 - Change notification
 - Frequent polling

Compare with synchronous tests that aim to fail fast.

- **Timeout is Failure:** fail the test if success not detected by some time limit.
- **Hide the Mechanism:** hide the synchronisation mechanisms behind abstractions that let tests express *what* is being tested, not *how* it is being tested.
- **Assert, Not Query:** avoids race conditions and ad-hoc synchronisation: you never know when you can query a value without polling for an expected state.
- **Single Timeout Definition:** can easily tune the timeout as the system changes to ensure that tests are reliable.

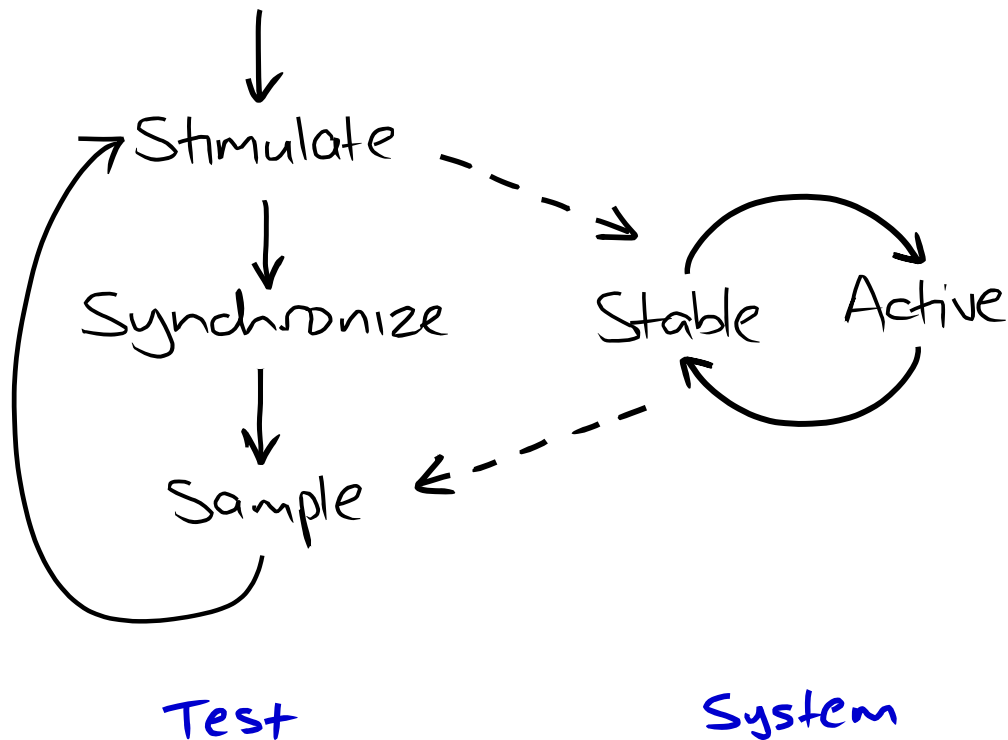
Polling & vs. Notifications



Notes:

Most significant difference between Polling and Notification. When polling, the test framework can miss changes that revert the state of the application. A test that collects notifications will see a notification of the new state and another notification of the change back to the old state.

Avoiding Lost Updates



Notes:

To avoid lost updates, a test must check for changes of state. It must stimulate the system to make it change state, then synchronise with the system by waiting for it to enter the expected state. It can then sample the state of the system, knowing that it will not change state while the sampling is happening.

We combine the synchronise and sample steps into assertions to hide the synchronisation mechanism away from the test code and ensure that the assertions Succeed Fast so that tests do not run too slowly.

In practise, we have not found lost updates to be a problem except when a system performs spontaneous behaviour (see below). Programmers naturally write tests this way, and so avoid synchronisation problems.

However, *Runaway Tests* are much more likely.

Polling and Runaway Tests


scan cheese @ £1

scan cheese @ £1


scan discount voucher @ -£1

assert displays total = £1

need assert
here to
synchronise



will
pass
at first
scan



Notes:

A *Runaway Test* synchronises on the wrong event, and so runs ahead of the system it is testing.

As a result, the test does not really test the behaviour it appears to, and will pass when the system contains a defect.

Programmers often write Runaway Tests when they issue a sequence of asynchronous stimuli to set up a test scenario.

Testing That Something Does Not Happen

cheese is in a 3-for-2 deal
scan cheese @ £1
assert total = £1
scan cheese @ £1
assert total = £2
scan cheese @ £1
assert total = £2

scan milk @ 50p
then assert
total = £2.50

← no change so
will pass
immediately

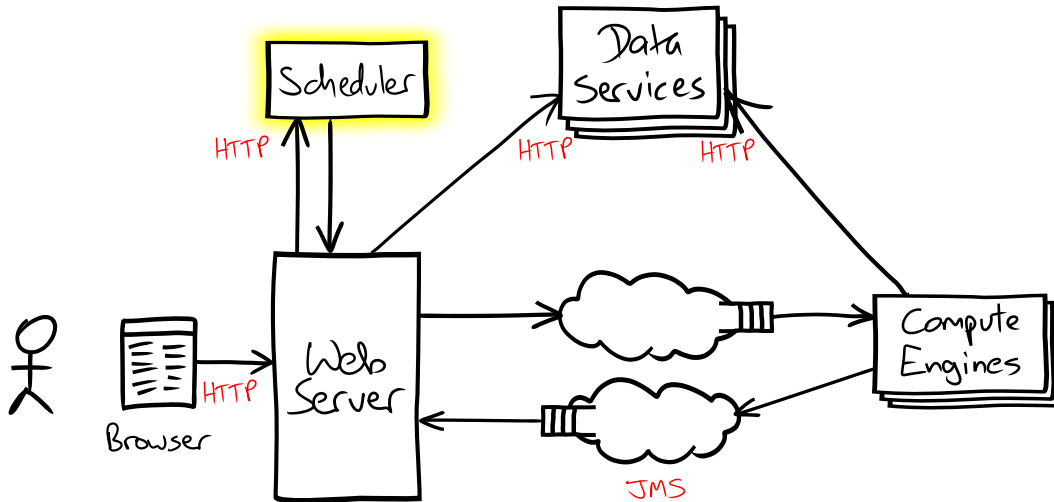
Notes:

Whether polling or recording notifications, testing that something does *not* happen is not as easy in an asynchronous test as it is in a synchronous test.

Naively asserting that the state has not changed will cause false positives: the test will run ahead of the system and not detect erroneous behaviour where the system state does change unexpectedly.

Must add additional test inputs that are processed sequentially *after* the input that has no effect, wait for that additional input to be processed successfully and then test that the undesired effect did not occur in between the two.

Externalise Spontaneous Behaviour



Notes:

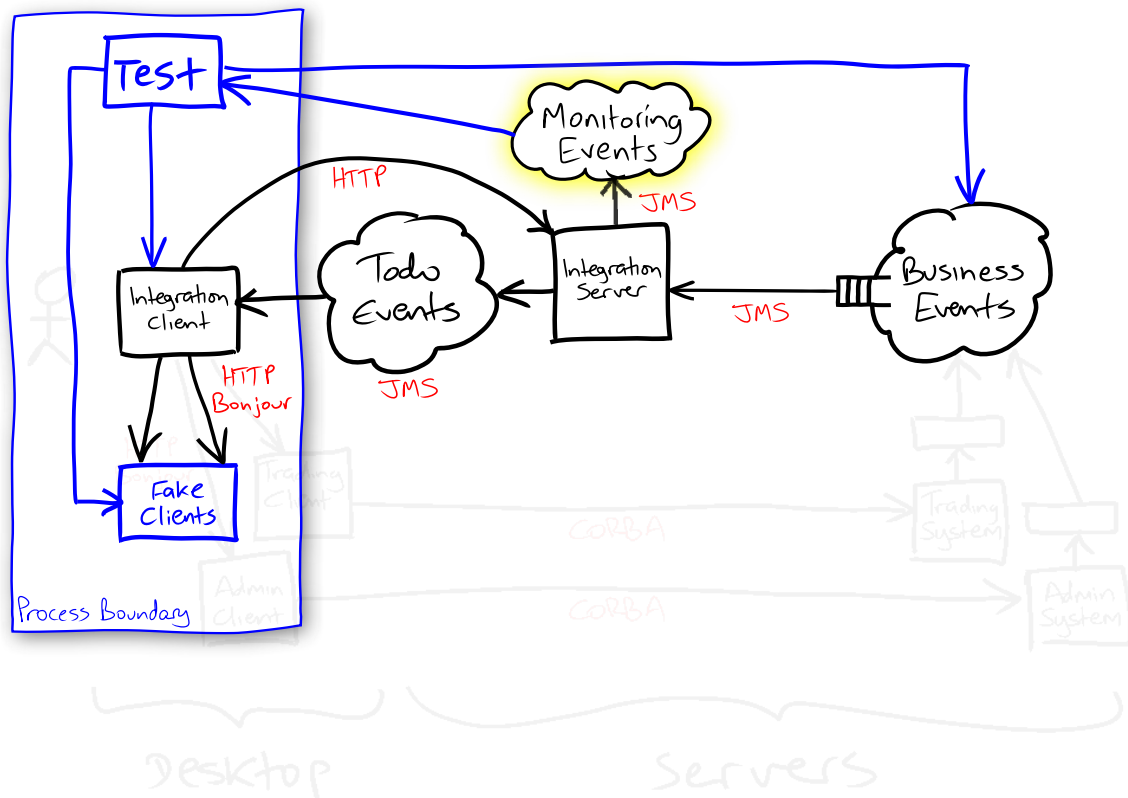
E.g. Scheduled activities. Treat the scheduler as a third-party service.

Avoid race conditions caused by state changes outside control of the test.

Allow tests to run as fast as possible by removing delays from inside system.

But, must test the external scheduler very heavily and ensure it has a very simple API/protocol, because it is not covered by system tests.

Add Synchronisation Hooks



Notes:

If a system is large it can be too expensive to start up the entire system and its underlying communication infrastructure for each test. You must therefore be careful that tests do not interfere with one another.

This occurs most often when a test fails part-way through. It also occurs when tests check only part of a processing sequence -- that various requests are not rejected by the system, for example.

In both situations, The system will continue to process any outstanding requests it has sent, and the results of that processing can interfere with the next test to run.

One solution is to add hooks to the system to let tests synchronise with its behaviour.

Hardware analogue: test ports on a circuit board.

For example, the "Send to Topics, Receive from Queues" idiom lets a test easily snoop on traffic between system components.

Both Loan-o-matic and Risk-o-matic publish a monitoring event on a topic as part of each transaction. This event is visible to other processes only when the transaction completes. This lets a test correlate request messages received from topics with transactions consuming those events. The test runner can wait for the system to become *quiescent* (when all requests, including requests caused by the processing of requests, have been fully consumed by the system) even when a test fails part way through.

In my experience, these hooks are often useful for monitoring the behaviour of the application in production, so Design for Testability also improves manageability.

Separate Concurrency Policy from Logic

```
Mockery context = new JUnit4Mockery();
AuctionSearchConsumer consumer =
    context.mock(AuctionSearchConsumer.class, "consumer");

StubAuctionHouse auctionHouseA = new StubAuctionHouse("houseA");
StubAuctionHouse auctionHouseB = new StubAuctionHouse("houseB");

DeterministicExecutor executor = new DeterministicExecutor();

AuctionSearch search =
    new AuctionSearch(executor, list(houseA, houseB), consumer);

...

@Test
public void doesNotAnnounceEmptySearchResults() throws Exception {
    Set<String> keywords = set("keywords");

    final List<AuctionDescription> auctionsA = list(auction("1"), auction("2"));

    auctionHouseA.willReturnSearchResults(keywords, auctionsA);
    auctionHouseB.willReturnSearchResults(keywords, noResults());

    context.checking(new Expectations() {{
        oneOf(consumer).auctionSearchFound(auctionsA);
        oneOf(consumer).auctionSearchFinished();
    }});

    search.search(keywords);
    executor.runUntilIdle();
}

... helper methods ...
```

www.jmock.org/threads.html

www.jmock.org/gwt.html

Notes:

A test that must deal with asynchronous code is slower than a test of plain old synchronous code. Threads, timeouts, synchronisation, messaging, etc., all take their toll.

It is hard to cover error paths because injecting faults is difficult.

So, synchronous unit testing is necessary even when you have a good test rig to help you cope with an asynchronous system.

Unit testing tools do not work well with asynchrony, but you can extract concurrency concerns from other logic.

The objects you want to unit test should not start threads themselves. Instead give them an object via which they schedule concurrent activities.

You can use a deterministic, synchronous implementation in unit tests and an implementation that spawns threads in the real system.

You can then write unit tests in the way you're used to.

The real implementation can be tested with all the techniques we've just looked at.

JMock provides deterministic implementations of Executor and ScheduledExecutorService in the org.jmock.lib.concurrent package.

More Information

www.natpryce.com

windowlicker.googlecode.com

www.mockobjects.com/book

Notes:

More information is available at the sites listed above.