XP Day 2008

Nat Pryce - nat.pryce@gmail.com

Version: 1.0
notes
Course code: xpday-2008

XP Day 2008

Build: 20
Built at: 2008-12-12T21:18:38Z
Course code: xpday-2008

Contents:

XP Day 2008..4

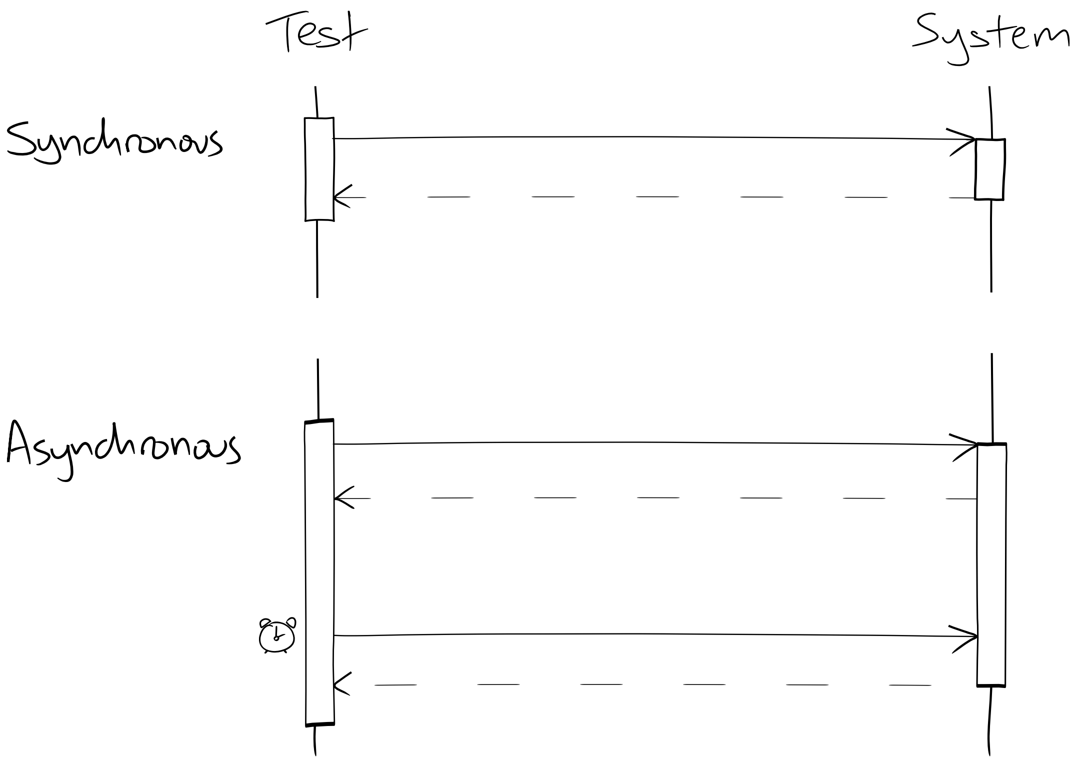
Unit 1

XP Day 2008

1. Test-Driven Development of Asynchronous Systems

Test-Driven Development of Asynchronous Systems

Synchronous vs. Asynchronous Tests



Notes:

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

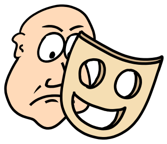
Synchronous test: control returns to test after system under test completes. Errors are detected immediately.

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

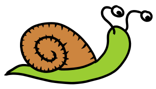
Failure is detected by the system not entering an expected state or sending an expected notification within some timeout.

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.

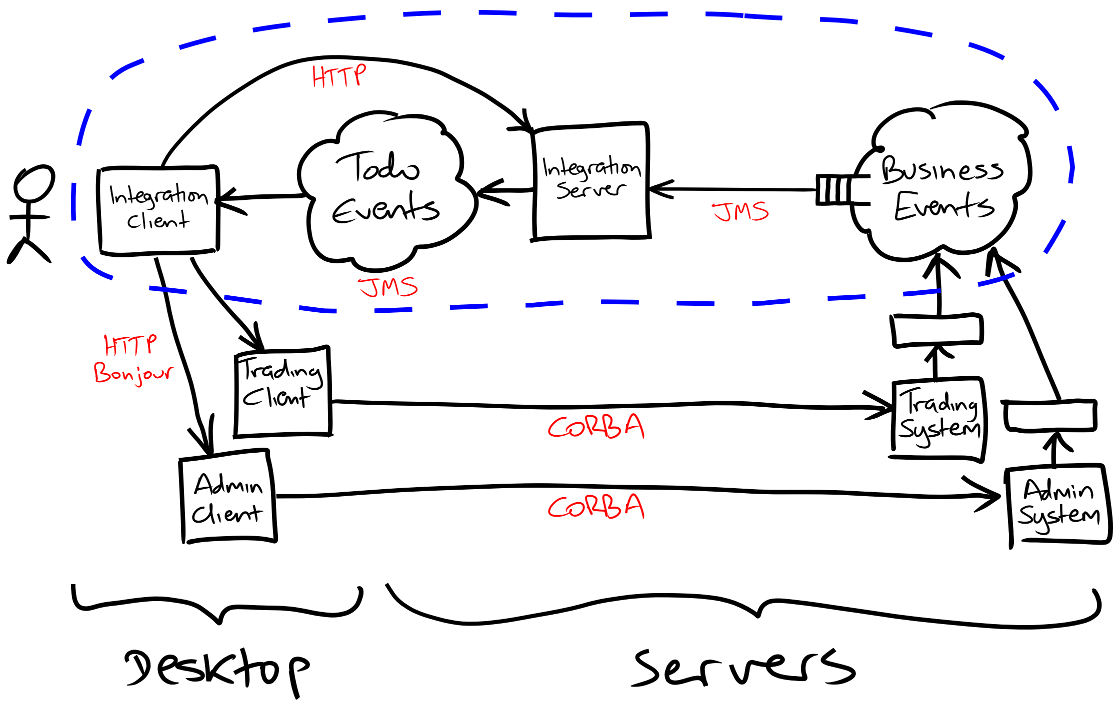
False Positives: the tests pass, but the system doesn't really work.

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 X



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.

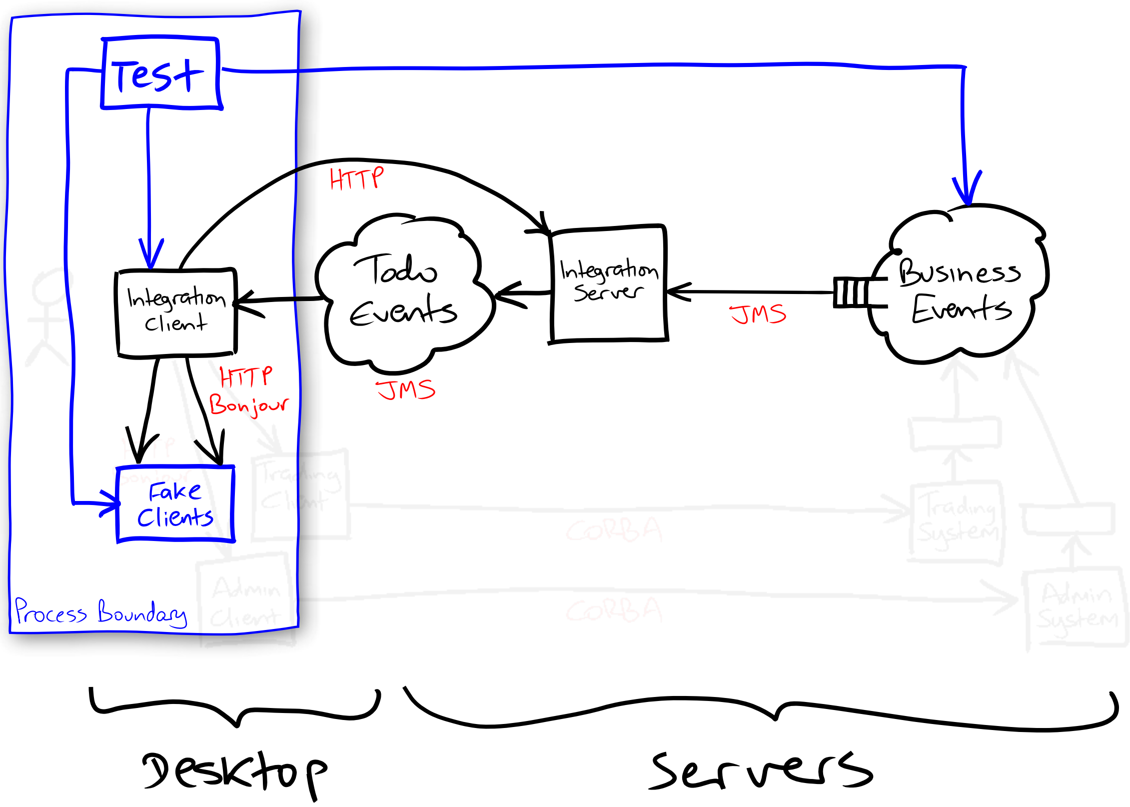
Our "to-do" system deploys "agents" that monitor the two systems. When an agent detects a significant change in the system it publishes a "business event". The server interprets business events by creating, modifying or deleting to-do items.

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.

Testing System X



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 themselves 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 the application's concurrent, event-driven architecture, we got GUI tests that coped with Swing's concurrent, event-driven architecture for free!

Example Test of System X

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

    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:

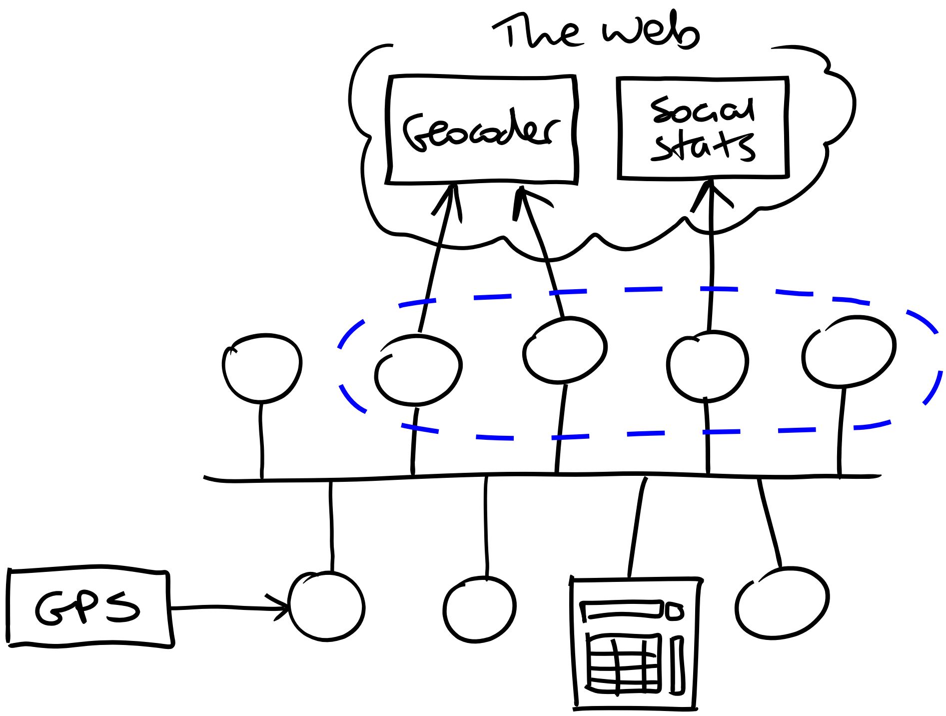
Moved all synchronisation out of the test classes, into reusable abstractions. Base test class defined common instances required for all tests and syntactic sugar.

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 Y



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.

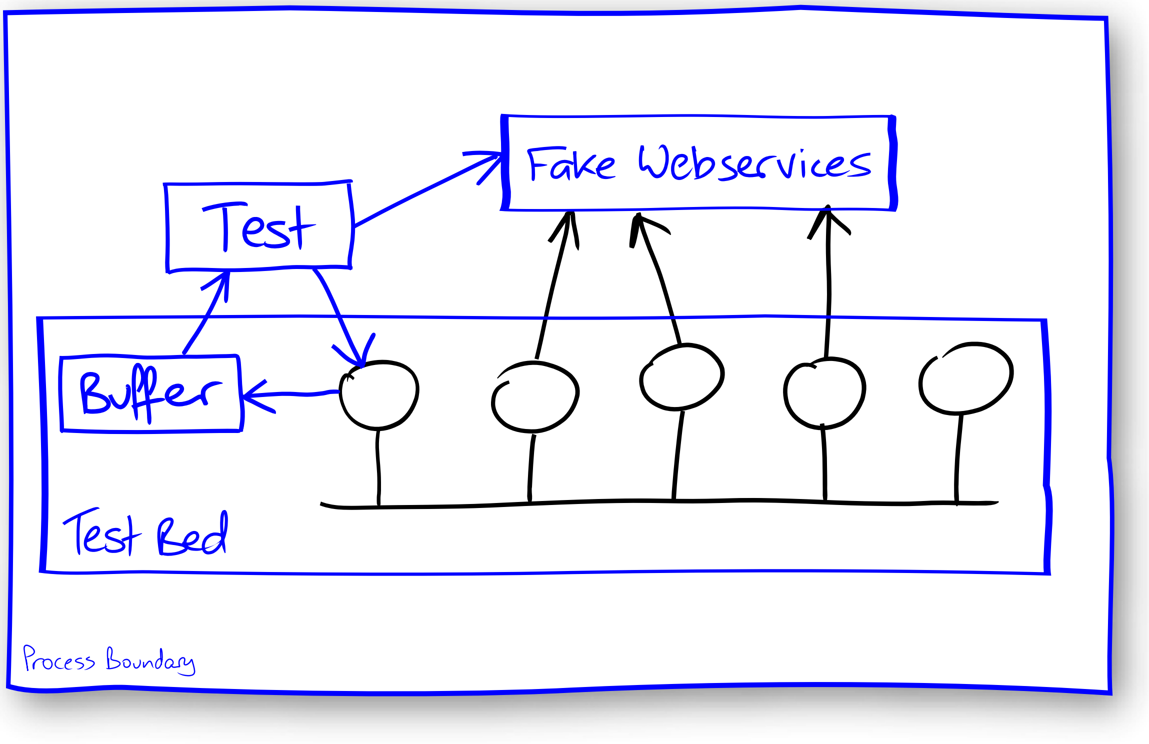
Components publish data read from hardware devices, translate primitive events into events with richer semantics, and consume events to act upon, display or record information.

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 System Y



Notes:

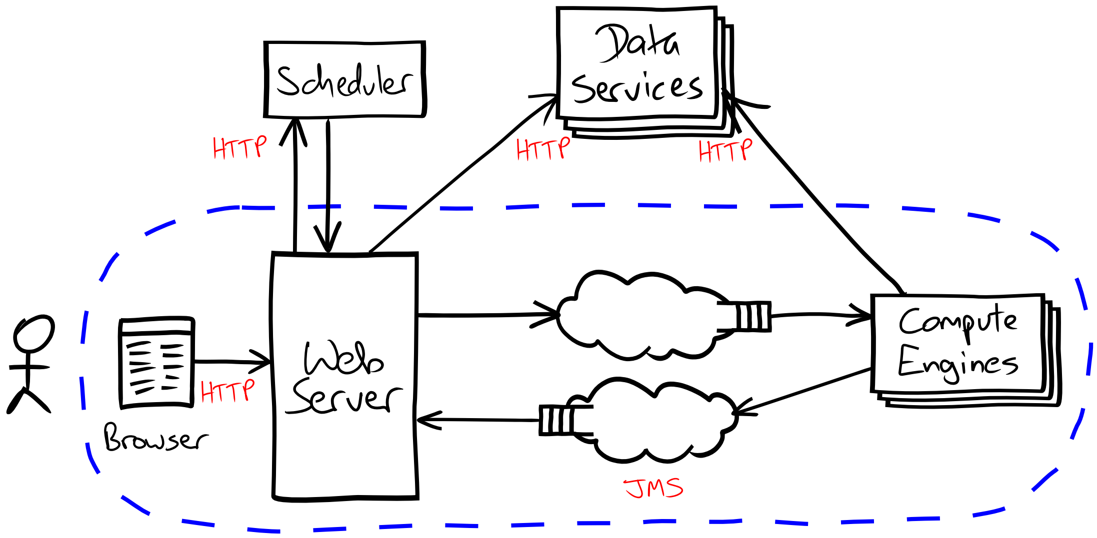
Testing a service is done with a "test bed", which is instantiated by a JUnit test.

A test bed contains an in-memory event bus for speed. The 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 Z



Notes:

System for performing complex financial calculations on a grid.

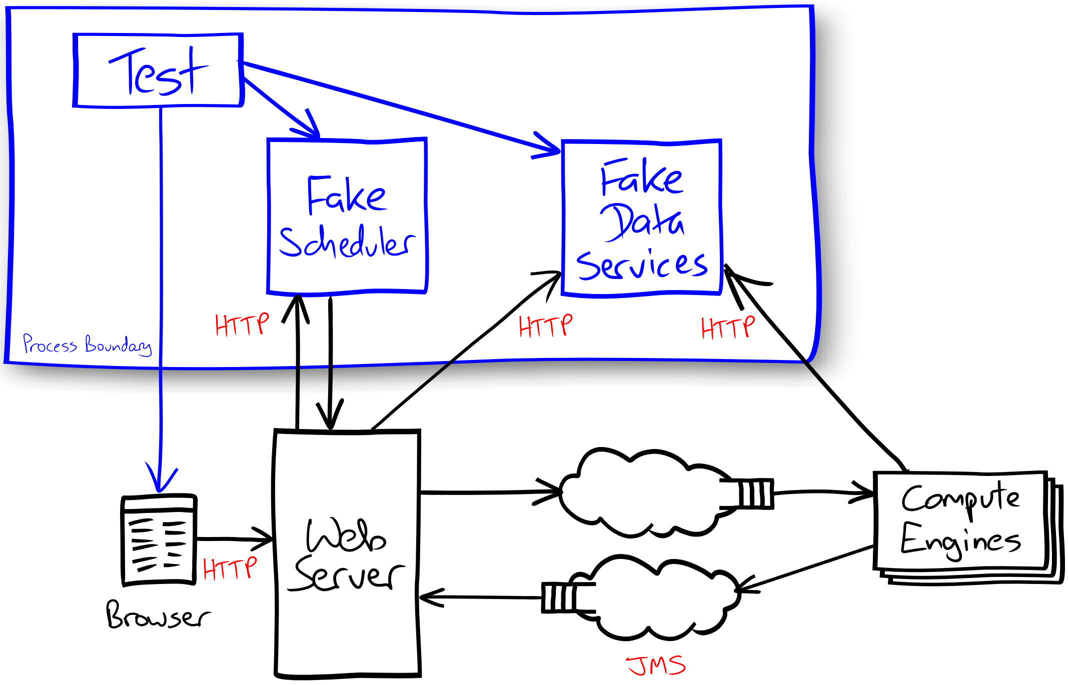
AJAX front-end written with GWT. Lets users define and schedule calculation jobs or execute jobs immediately.

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 System Z



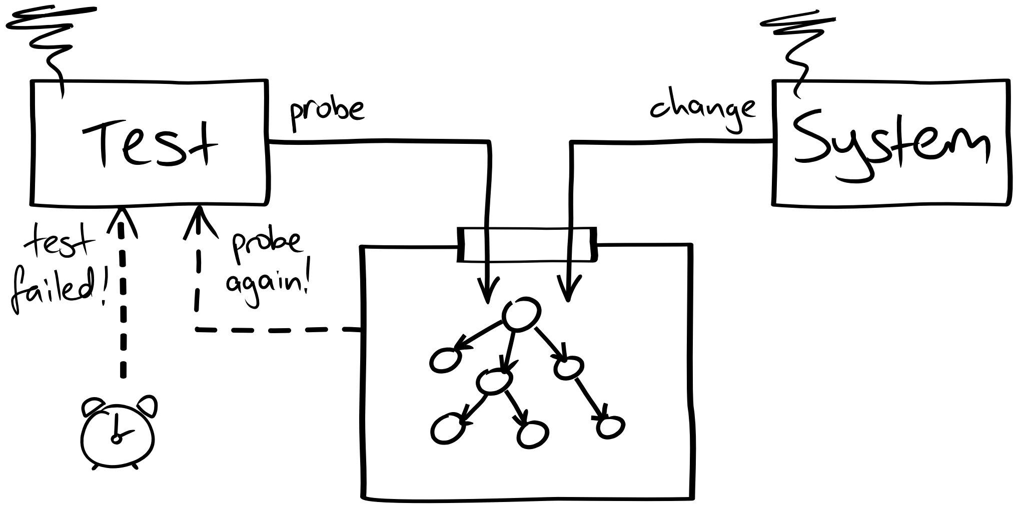
Notes:

Similar to System X, 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

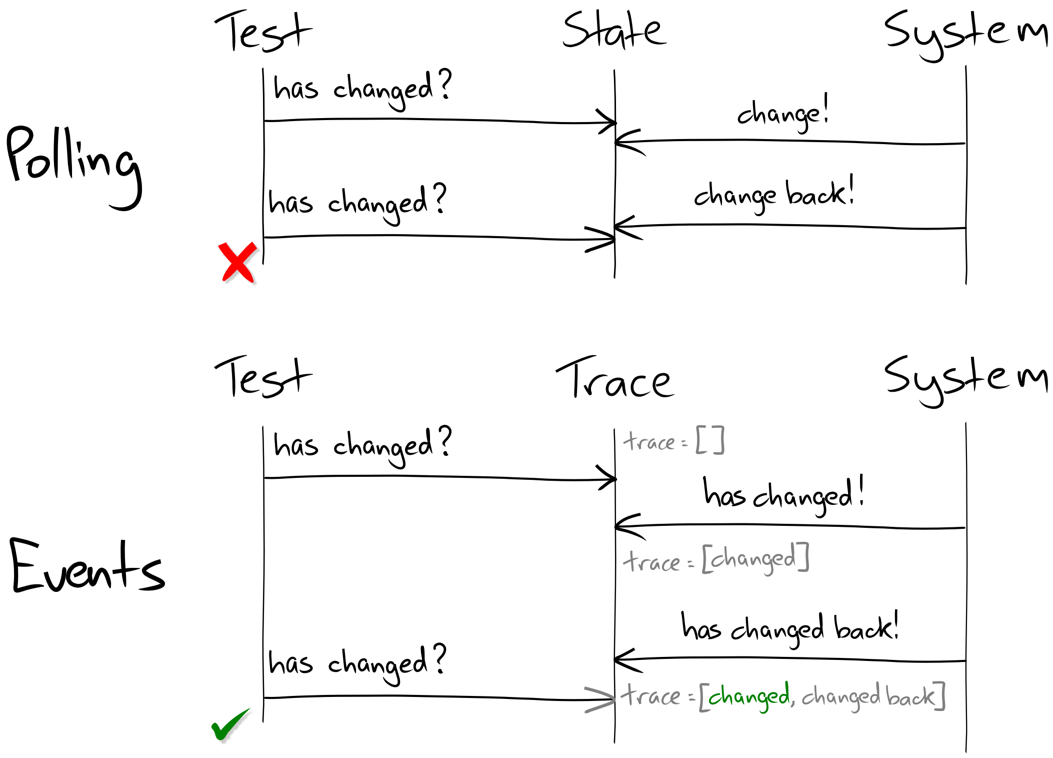


Notes:

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
- **Timeout is Failure**: fail the test if success not detected by some time limit.
- **Hide 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.

Polling & vs. Notifications



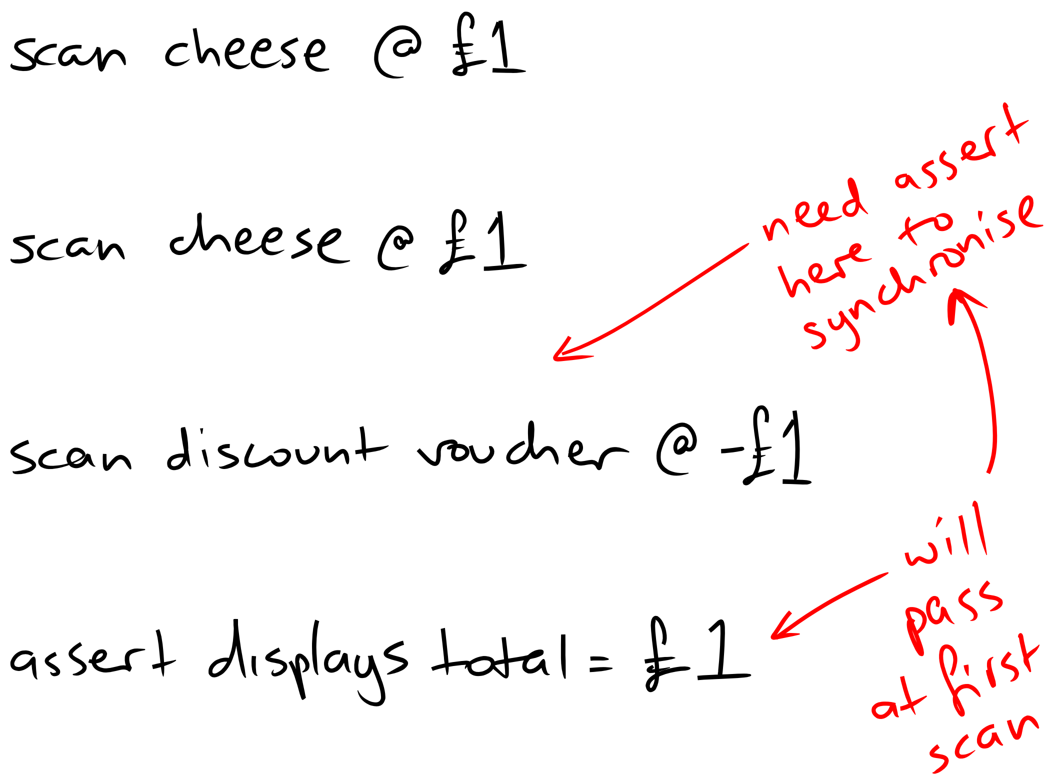
Notes:

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.

In practise, we have not found this to be a problem except when a system performs spontaneous behaviour (see below).

However, *Runaway Tests* are much more likely.

Polling and Runaway Tests

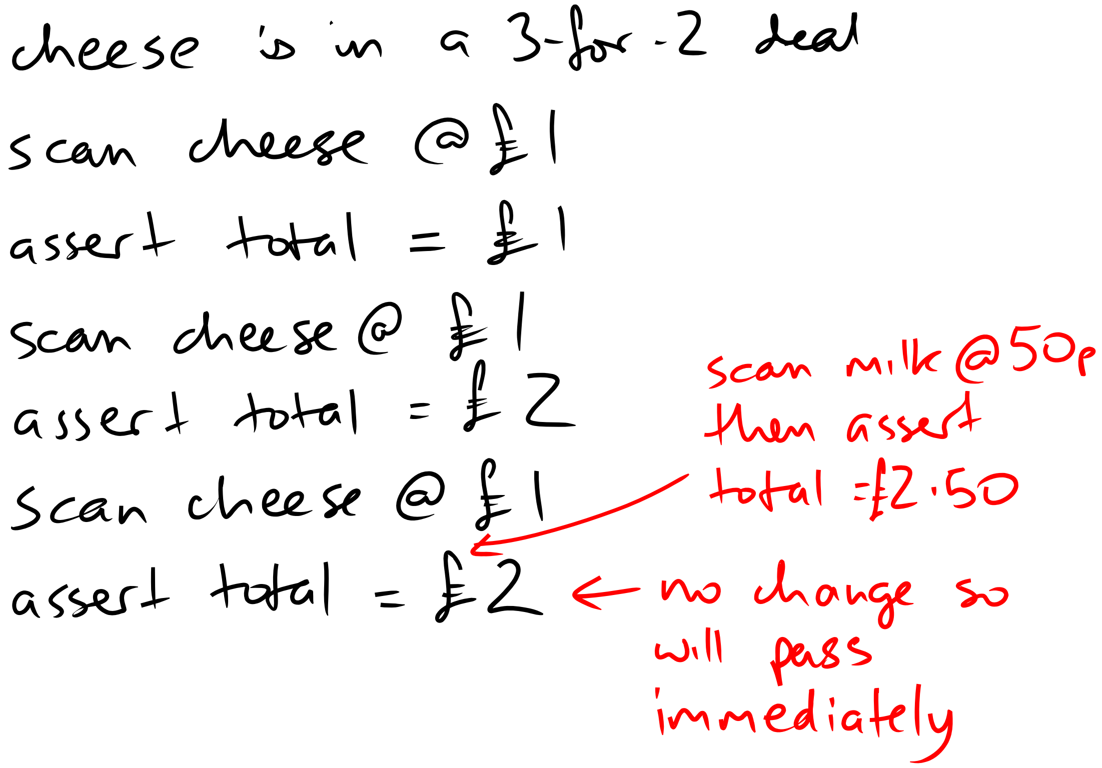


Notes:

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.

Testing That Something Does Not Happen



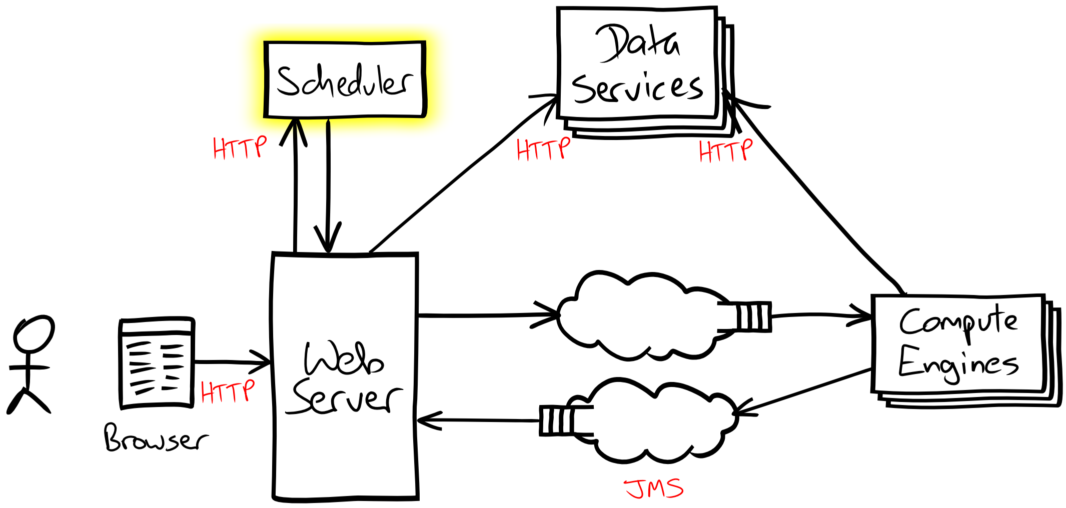
Notes:

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 w.r.t. the input that has no effect, wait for the that input to be processed successfully and then test that no unexpected behaviour occurred in between.

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.

More Information

www.natpryce.com

windowlicker.googlecode.com

www.mockobjects.com/book

Notes:

More information is available at the sites listed above.