Datatracker Testing

Making the users happy
by catching bugs

Contents

- How test coverage has changed
- Catching data corruption
- Finding user points of pain
- Dead code removal
- Chasing the holy grail of full test coverage

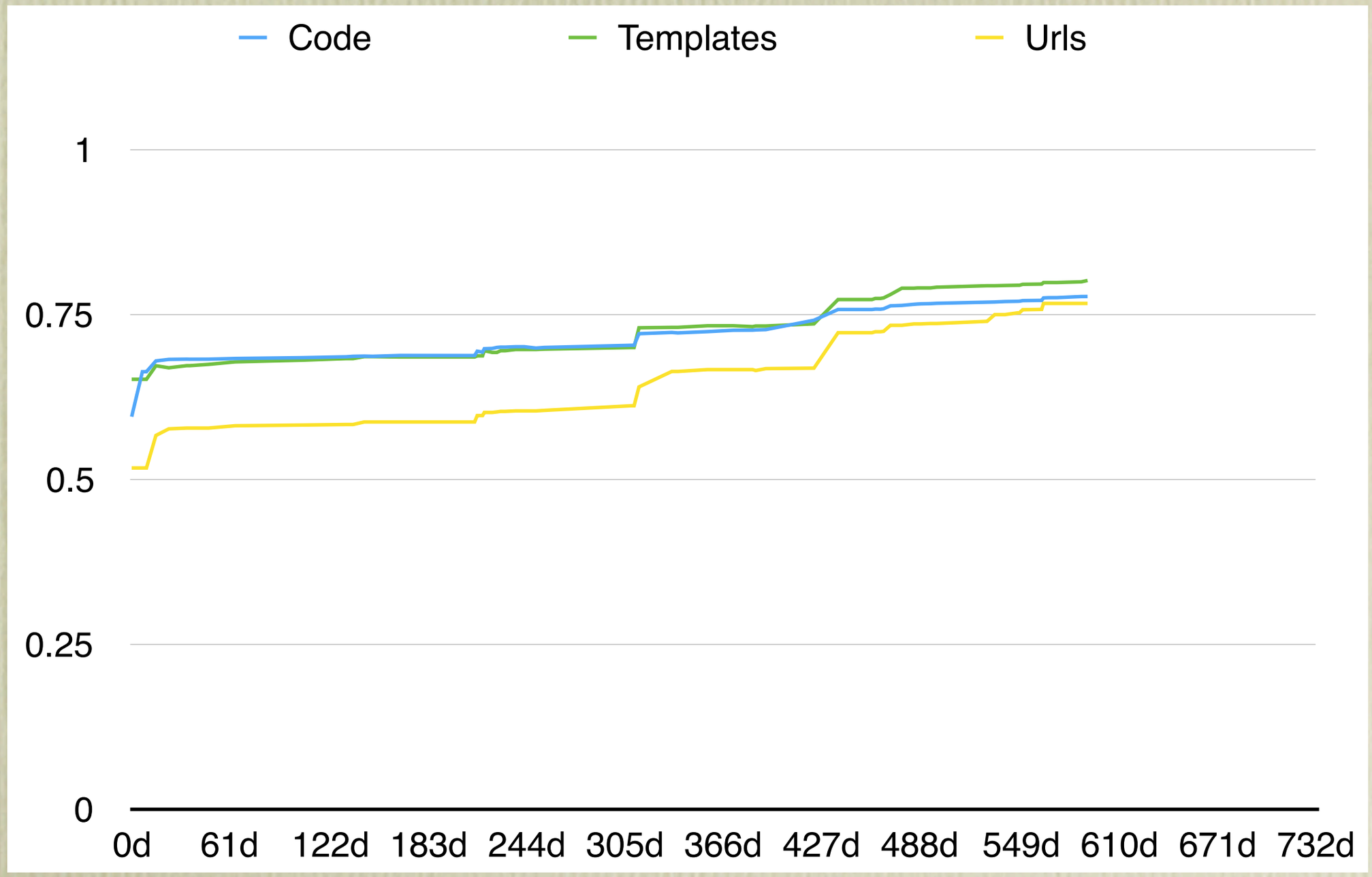
What you measure is what you get ...

Test coverage measurements

In March 2015, we started to measure test coverage (code coverage, template coverage, and URL coverage) as part of the test suite.

We also made the test suite fail if new code had less test coverage than that of the latest release, ensuring that the coverage could go up, but not easily down.

Test coverage since measurement start



However ...

All bugs are not
of the same nature

Bug: Document history with wrong event dates

We recently had a case where new code added events with the wrong timestamp to a document's history.

While it was fairly easy, in that case, to fix up both the code and the database entries, it points to a class of errors that unit tests aren't good at discovering.

Unit tests are normally written to confirm that the code is doing what the programmer believes is correct behaviour for a given page or function, not to confirm general expectations about data structures.

Bug: Meeting sessions with multiple time slots

In another recent case, we had some sessions that somehow had been associated with multiple time slots, something which should never happen.

Again a case of bad data in the database, fairly easy to fix. It is however hard to know just where and when the bad data was introduced, and hard to write unit tests to catch this.

Programming by Contract

Some languages have explicit support for Programming by Contract (or Design by Contract), which is a way to make sure that certain pre- and post-conditions always are fulfilled when entering/leaving a function or method. The term was introduced by Bertrand Meyer, with his language Eiffel.

The idea is closely related to the idea of **invariants**, which is quite a bit older, and promoted by for instance Niklaus Wirth (of Algol, Pascal, Modula-2, Euler, Oberon languages fame).

Checking invariants, pre- and post-conditions

Python is not designed with specific support for programming by contract, but it has the `assert` statement, which lets us write code that gives an exception when we violate a pre- or post-condition or an invariant.

Assertions are not meant for use in handling of bad data, bad user input or the like — they are there to catch programming **bugs** close to the point in code where the bug is, and close in time to when the buggy code is executed.

Fast and slow assertions

Some invariants can be very fast to check. In this case, the Python `assert` statement, which remains active in production code, would be the right thing. If we had had a suitable assertion about event timestamps, we would have caught that error early and would not have added bad event timestamps to the database.

Other invariants can be much slower to check. In this case, a suitably designed function, say `debug.assert()` which is active only during development seems best. It doesn't offer the same level of protection, but should catch invariant violations during development testing.

Hard and soft assertions

Another dimension is between genuine asserts, which raise an exception and thereby stop execution of the regular program flow, and 'soft' asserts, which only note, log, or otherwise communicate the assert failure, without halting the regular program flow.

It can be argued that the wrong event dates, since they can be fixed later, should not stop regular program flow, but just notify people about the failure to maintain the invariant. This would let the draft submission go forward, even with the bad event dates.

Proposal for use of assertions

We start to use `assert`, `debug.assert()` and `soft_assert()` statements in the code, and add a test to the test suite which measures the ratio between number of asserts and lines of code, and warns (not fail) if the ratio is dropping.

We cannot however use quite the same approach as for test coverage, where the test suite always fails if the coverage drops, forcing the coverage always upwards: 100% coverage is good, but 100% assert statements is not. On the order of 1 statement for every 50 lines of code seems a reasonable ambition.

Measurements

Points of pain

How can we most effectively improve responsiveness? (i.e., which often used pages are slow and should be speeded up?)

Which pages are frequently used, and should be available with few clicks — but are not?

Are there pages which are often used together, which could be rearranged to provide all information on one page?

Measurements

There exists a number of Django apps which does in-production measurements which should let us answer those questions, and more, in order to improve

- Responsiveness
- Navigation
- Page composition

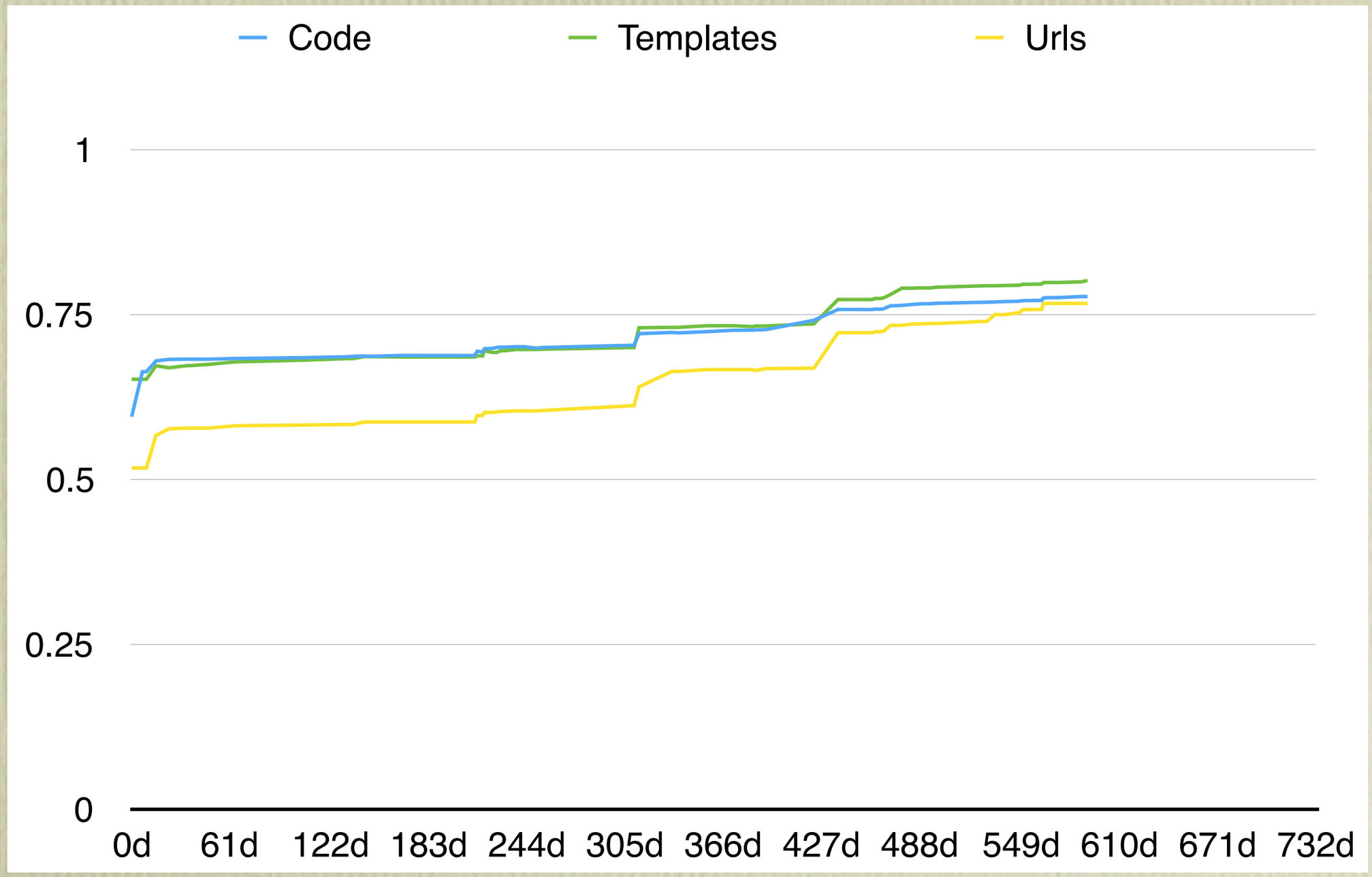
Should we explore these, and start using one or more, in order to collect data that will let us improve things?

Proposal for measurements

Spend some time (a man-week?), before we get to the point of doing serious performance improvement work, in assessing and trying out measurement apps for Django; and then settle on one or more and integrate it.

Dead Code

Test coverage since measurement start



Diminishing Returns

As we continue to add tests to the test suite, we're seeing diminishing returns in total test coverage — even if the new tests cover the new code, we're sometimes seeing that it gets harder to push the total coverage noticeably upwards. This is expected.

However, it also brings us this: I've recently examined some modules which had good coverage, but were not at 100%, and found a very nice thing:

We're now getting to a point where the test coverage is good enough to be useful in showing dead code :-)

Marking possibly dead code

In addition to using the code coverage measurements to identify possibly dead code, and regularly come back to inspect and remove such code, it might be useful to have a way to mark possibly dead code, in such a way that subsequent execution of the code would be noticed and the dead code assertion would be refuted.

We could have a `@dead_code` decorator, and a `dead_code()` function, which would log any execution in order to refute the assertion that the code is dead. This would make it easy for sprint participants to mark code as possibly dead, for later removal.

Proposal for cleanup

Let's add the `@dead_code` decorator/function, and put an activity on the list of maintenance tasks which is to go over modules with code test coverage above about 80% and see if the parts without coverage are actually dead code that can never be reached by our tests because it isn't used in rendering any pages.

We could schedule such an activity right after regular Django version upgrades; that would give us a reminder now and then to do a scan for dead code, and it would happen at a point where one had already been all over the code looking for upgrade issues.

Testing the Test Coverage

Testing Code Test Coverage

Today we measure code test coverage as a per-file percentage and an overall percentage. This lets us fail the test coverage test if the overall percentage goes down, and it has served us well in increasing coverage.

However, what if new or changed lines of code were directly checked for test coverage, and if not covered would cause a test coverage test to fail? We would know immediately which changed lines need tests.

The Python `coverage` module emits per-line coverage information, so creating this kind of test is feasible.

Test-Driven Development

This would to some extent promote Test-Driven Development (TDD), as it would encourage writing the tests first, so as to have successful tests as soon as the code works.

It would also encourage tests which exercised all paths through `if` statements, something the current percentage-based test coverage test doesn't do.

Proposal for changed testing of code coverage

Add a new code coverage test which checks the code coverage of modified lines of code, reporting whether new and changed lines are covered by tests.

This would not replace the current test, but add another way to point out places where test coverage is missing.

Make this advisory to start with, rather than a test which fails if new or changed lines are not covered by tests. We can change this to a test which fails on missing coverage later, if desired.