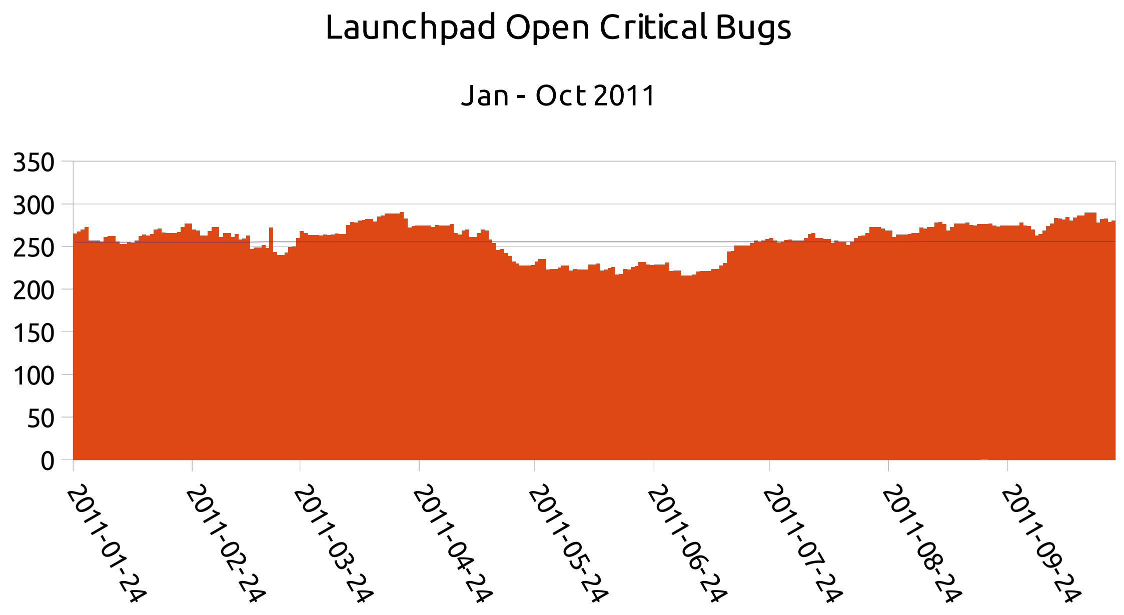
Launchpad FY2011 Q1 & Q2 Critical Bugs Analysis

Introduction

That analysis was prompted because the long-term trend around our Critical bugs pool is flat (y=252). We roughly get as many new criticals as we fix. This hint at that we have serious quality issues and that we need to change things in order to quench the source of new criticals.



The Data

For Canonical folk, you can retrieve the original spreadsheet containing the data used in the analysis is available in the Launchpad FY2011 Critical Bugs spreadsheet.

For others, you can follow along on the published version.

You can switch output=xls, output=ods, or output=csv to retrieve this in a machine-readable format.

Methodology

I randomly selected 50 bugs from the 389 that were filed since April 1st. (I selected April 1st because I thought that we were done with legacy issues and that would expose quality issues in _new_ development).

(To random sample, I use =RANDOM() and then sort the bugs and selected the first 50, after removing Invalid bugs and the one from the results-tracker.)

I then look into each bug, reading the related merge proposals to find where the bug was introduced.

Several columns were filled during that process:

- **Comments**: Usually a text description of where the bug was introduced, or what caused the bug to be introduced.
- **Root Cause**: A category tag attempting to group related kind of problems together.
- **Intro Revision**: The source code revision that should first contain the bug. This isn't always filled. It should be filled for all bugs introduced recently though. When it became obvious that the bug was a legacy one (introduced before the squad reorg), I didn't bother trying to pin-point the exact revision.
- **Intro Rotation**: The rotation type (feature, maintenance) that was responsible for introducing the bug. legacy is used for issues dating from before the squad reorg. (community and thunderdome are also two rotation-types you'll see used in a few cases)

Three fields were filed automatically based on the closing date and the assignee:

- **Squad**: The squad that fixed the bug.
- **Rotation**: The type of rotation on which the squad was at bug fix time.
- **Feature**: When the rotation is feature, it contains the feature the squad was working on at the time.

Results

Two pivot table were created to summarize the data.

Introduced vs Fixed

	unfixed		**community**		**feature**		**maintenance**		**support**		**Grand T**
unknown	1	2%					1	2%	2	4%	4
community			1	2%							1
feature					6	12%			1	2%	7
legacy	14	28%			1	2%	13	26%	1	2%	29
maintenance	2	4%					3	6%	1	2%	6
thunderdome	1	2%			2	4%					3
Grand Total	18	36%	1	2%	9	18%	17	34%	5	10%	50

This table shows in rows the rotation-type that introduced the bug and in columns the rotation-type that fixed the bug. From that table, we can see that in our sample:

- 58% of the new bugs are really legacy issues that were not reported
- 36% of the bugs were not fixed during the period
- almost all new criticals introduced by a feature squad were fixed by that same squad (6 out of 7)
- maintenance rotation fixes the bulk of the legacy issues (13 out of 15, 14 being on the backlog still)

– feature and maintenance rotations introduced about the same number of new criticals

– ~25% of the new criticals is the result of new work (feature or maintenance)

Root cause by source

			community		feature		legacy		maintenance		thunderdome		Grand Total	
critical_by_inheritance	1	2%											1	2%
data_corruption							1	2%					1	2%
deployment_interaction					1	2%	3	6%	1	2%			5	10%
development_production_drift											1	2%	1	2%
failed_to_handle_user_generated_error							1	2%					1	2%
flaky_tests							2	4%					2	4%
incomplete_refactoring							1	2%					1	2%
insufficient_scaling					2	4%	10	20%					12	24%
invalid_priority	2	4%											2	4%
leaky_abstraction					1	2%							1	2%
merge_conflict					1	2%							1	2%
missing_integration_test							2	4%	3	6%	2	4%	7	14%
missing_interface_test			1	2%			1	2%					2	4%
missing_unit_test					1	2%	4	8%					5	10%
open_transaction							1	2%					1	2%
requirements_misunderstanding					1	2%							1	2%
script_db_permission							2	4%	1	2%			3	6%
unexpected_interaction							1	2%					1	2%
unknown	1	2%											1	2%
view_model_validation									1	2%			1	2%
Grand Total	4	8%	1	2%	7	14%	29	58%	6	12%	3	6%	50	100%

This table shows in rows the class of bug and in columns the rotation-type that introduced the bug. From it, we can see that in our sample:

– (worth highlighting again) 58% of the bugs are legacy one

– a quarter of the bugs are related to insufficient_scaling – these are all performance problems in our model. That also represents nearly 33% of the causes of the legacy bugs

– 28% of bugs would have been prevented by proper testing, either at the unit-test level (missing_unit_test: 10%), interface (missing_interface_test: 4%) or integration (missing_integration_test: 14%)

  – missing_unit_test represents bugs that wouldn't have occurred if a strict TDD-mode would have been used

  – missing_interface_test represents bugs that happen when an interface is either extended or implemented but there is no test for that contract, so change in the required contract are not caught

  – missing_integration_test represents both case where the business rules we need to support wasn't tested (nor documented) as well as cases where it's the integration between two components that wasn't tested

- 10% of the bugs were really deployment interaction issues. These are cases where an OOPS or another error only occurs during deployments
  - either because the code doesn't handle that kind of interruption like bug #810694
  - or because during that time different code version were running at the same like in bugs #779489, 810116 or #820389.
- another substantial source of criticals is our policy of very fine grained database permissions for scripts which represents 6% of the sample
- 8% of bugs in our sample were not critical per se
  - 4% didn't match any of our critical criteria (invalid_priority)
  - 2% were only critical because it was considered a dependency of another real critical (critical_by_inheritance)
  - 2% (that's one bug) fixed itself and the reason couldn't be discovered (unknown). That was bug #751202.
- about another quarter of the sample represents various category of errors for which there was only 1 or 2 bugs in our sample

Recommendations

Based on these results, I can think of the following recommendations.

Launchpad has a lots of legacy

This is not a recommendation, it's more the summary of the background in which we are operating: really, Launchpad is a legacy application. 58% of the new bugs were actual legacy bugs! That means that have probably hundreds of bugs matching our critical criteria lurking in various part of our code base. The inconsistency in our test coverage means also that we are likely to introduce many more bugs than we should in the future. (of the 14 bugs related to spotty test coverage, 7 were legacy and 7 were part of new work).

Given this I think that focusing on performance and testing are likely to be our two best strategy to reduce the incoming flow of criticals.

We are not done with performance

The single most specific cause of bugs were performance related. So we should renew our efforts to improve our model to make it performant. Performance bugs are not trivial to fix. In many cases, we'll either need to rework the model to make it possible to operate on sets instead of individual objects (the death-by-a-thousand-sql-queries problem) or rework the schema. Fortunately, it's not easier than ever to iterate on schema changes. And we have good patterns for the operate-on-set case.

So I would recommend that maintenance squads should focus on timeout related issues first and foremost.

This will have compounding effects:

– it will improve the live of users, always a good one :-)

– as we rework the model to operate on sets or introduce faster schema, it will be easier for the next person, to build on that and make other things operate on sets

– people on maintenance will build their awareness related to performance and will probably mean they will be less likely to introduce performance regressions while working on features

So while working on maintenance rotation, a good rule of thumb could be ask yourself: in the bugs I fixed recently, was there a performance-related issue? If not, please pick one as your next target.

Test or die

We can do better at our testing story. Missing tests represent the other big source behind new criticals. In this category, I would suggest things along the following lines:

– if you are not writing your code TDD-style, start doing it, really

– when reviewing code, be very nosy about test coverage

– if you are extending or modifying a component, make sure there is an interface test, if there isn't one, write one

– we might want to investigate acceptance-test-driven-development (Fitnesse style) to make sure that our business rules are well tested and documented. The Robot Framework seems like nice fitness suite for Python.

With the LaunchpadObjectFactory, it's now easier then ever to write good unit tests, so there is no reason we shouldn't be doing it. We also have much better javascript testing infrastructure than ever. The parallellisation of the test suite (will be started by Yellow once Orange completes Custom bug listings) should also help kill the meme that we shouldn't add too many tests because it slows down the test suite.

Deployment-related issues

We should probably remove the policy about fine-grained DB permissions for scripts. It's not giving us much, and it trips us very often. Let's use the same permission for scripts than the web app (while maintaining a separate user).

I'm not sure if there is anything we can do about the deployment interactions class of issue apart. We should learn to be more defensive when writing code and think about how this code will interact with older version of the code running concurrently. We will need to develop this awareness more and more anyway as we move to a SOA archicture. The fastdowntime also introduce more conditions of this kind. I don't have anything more concrete here than let's remind each other about this aspect.