Some notes Gnome Testing

Abstract

A short report on my work exploring the various methods of testing employed currently in the Gnome projects, the state of testing as a whole, and our current state of qa work. First a brief section about various kinds of testing, then how they are currently in use or not. Following up with a suggestion on how to improve the current state of things.

About the Author

I'm D.S. "Spider" Ljungmark, Long time Linux user, developer, administrator and general geek. Over the last year I've become more deeply involved in software testing; I decided to apply some more of this domain experience onto the Gnome ecosystem in order to see what I could learn and help out with.

Before this, I was a Gentoo packager for several years until retiring from the project, where I was also working. (spider@gentoo.org)

This article is a distilled form of my talks on the subject, held at Guadec, Spain, July - August 2012.

About software testing

Software testing is a relatively young field in terms of popularity, and has been developing rather rapidly during the last decade, especially as development models have changed, causing the testing process to change as well.

When naming things in "Software Testing" there's an abundance of confusion, disparity and methodologies available. I'll just go over a few of these.

Unit Tests

Unit tests are probably the most well known method of testing, and is generally well perceived. They are a programmer aid, at a low level of the code, testing the smallest piece possible for known good / known bad functionality. It is more a tool to prevent regressions when re-factoring and maintaining code, than a tool to increase an arbitrary term of "Quality".

The value of Unit testing vs. Code By Contract and strict syntax languages can be debated forever in academia, but it is generally recognized that unit tests increase in value as you add developers to a project.

Integration Tests

Integration tests come in (at least) two shapes, but are in all forms testing the edges and cooperation between projects and others.

In the simplest form, this is a linking level test, making sure that the application still builds against libraries after upgrades.

In the more advanced setting it is a protocol level test, where tests go against an external server in order to make sure that communication doesn't break. Integration tests of this kind usually belongs on the lower end of the protocol stack, in libraries defining HTTP/IMAP access, Dbus calls, and similar.

Smoke Tests

Smoke testing has it's name from the hardware world, where a unit was plugged into a power source. If blue smoke didn't happen, it passed smoke-tests.

In the software world it's usually centred around software starting after build, supporting something minimal.

Performance Tests (KPI tests)

KPI stand for "Key Performance Index" and they are a way of measuring that there are no regressions in performance. The testing is either done manually (stopwatch and button presses) or programmatically. Commonly they contain start-up, common actions, or load testing.

Example: measuring the time of an import job ran repeatedly; measuring the CPU load of a user login to a web forum. Me

Acceptance Tests

Acceptance Tests come in two forms, classical, and "Agile"

In Classical acceptance testing, it is the job where a client verifies a delivered product (Bridge, software, car) against the contract or work order, and is only performed at the end of the development cycle. It usually involves massive amounts of work and long hours of lawyers.

In the more modern "Agile" method of the same name, this is limited to testing the desired function of a single feature after completion. This usually also involves making sure that development process was followed (documentation, unit tests, check-ins) as well as ensuring that the user story was fulfilled according to spec.

Acceptance tests of this kind are usually Automated, while the first kind are a mix of automation and manual.

Functional Testing, System Testing

Functional Testing is what is considered when we ask if something meets design criteria. It is usually pitted against System Testing, which is asking if something meets User Criteria.

Both are commonly performed manually.

Gnome Method of Testing

The Gnome Method of Feature Testing was established in the 1.4 to 2.0 days, and reintroduced to popularity in the 3.0 release. It involves removing all traces of a feature from the code base, waiting for a release cycle and listening for complaints that it was removed and that Gnome developers aren't listening to feedback. If there are no complaints, the feature was broken and doesn't need to be fixed. If there are complaints, it's considered for redevelopment.

For more information about Software Testing, see http://www.satisfice.com/ or similar resources.

The Status and History of Software Testing in Gnome

Unit Tests

Unit Testing is commonplace in many OSS projects, and are what people mostly consider when they hear "testing". Various projects use it to differing degrees, but overall it's well considered, recommended and in use.

Integration Tests

Integration tests is done by several parties in the Development and Distribution cycle. Of the two flavours, a few projects have service type of integration tests, testing the ability to connect to localhost with sftp, imap servers, libsoup tests against remote servers. On the whole, this kind of testing requires a formal setup and laboratory with various kinds of servers to integrate against, something that few OSS developers have access to, and which would require the infrastructure support by a larger organisation to support. There is certainly some of this happening behind closed doors at various distributions, but this remains a hidden effort, if it exists.

The other kind of integration is an area where the Distributors today are doing a stellar job. Both testing new libraries against previously compiled software, as well as testing rebuilds of the whole software stacks to ensure source level compatibility. The job is spread out between many parties but with "upstream first" and similar incentives. A lot of work is being shared between parties. Trivial fixes tend to be solved on many places at

once, while more advanced fixes usually end up upstream with a "formal blessing". Of course, there have been rare cases of the procedure not working, though on the whole, this level of testing is a solved issue.

Smoke Tests

Smoke testing is basically done by a developer in the compile - launch - crash cycle, and is universally performed by developers as well as distributions. There are some notable exceptions such as Fedora's Rawhide (re: https://lwn.net/Articles/506831/)

Performance Tests

As a rule there are few performance concerns, and fewer tests still. The GTK+ toolkit has some, but there are no regular tracking of performance regressions (due to in large part, lack of continuous integration and associated quality reports)

Examples of regular KPI's that could be done, and maybe, should be done, would be "start-up until icons rendered" in Gnome Documents (currently, 8+ seconds at times) "Start-up time of Rhythmbox" (until it can select a song and play it) (varies depending on library size.), "nautilus browsing of /bin" or the time it takes Evolution to display a mailing list folder.

On lower levels, there are tests on performance of gstreamer encoding, decoding, as well as timing issues of certain unit tests, however there appears to be little concern for this overall, and no continuous reporting of progress, regression or stagnation.

Automated Acceptance Testing

There has been attempts in the past, with Mago (last 2.12 or similar?) Linux Desktop Testing Project and Dogtail.

Some applications (PiTiVi) have actively maintained acceptance tests using this/that work and are a core part of the current development, but as a whole this has been a largely abandoned and ignored part of development culture. The various attempts to organise such testing have stagnated as they are not part of the daily development.

Musings:

In order to have such tests working, they have to be part of the project and be maintained together with the project. This means being part of the build system (not building without testing infrastructure in place) as well as being part of the commit and release criteria. Staged commits, automatic reverts and a culture of accepting the fact that "breaking tests are bad" needs to work before we can enforce testing. Also, adding testing as an external project is doomed to obsolescence and breakage during the next development cycle. Making the tests a burden rather than a gain.

System Testing

There is no concentrated effort in Gnome at the moment to do this kind of testing. Various distributions have organised test days (Fedora, Ubuntu) test teams (Ubuntu, SUSE) which do manual testing with users as well as professional testers.

These tests are probably overlapping, and vary in quality and range of what is tested. All from "Create a new launcher in the dash" to "merge conflicts in file copying between Samba and local content."

Other than that, there is the well known Debian model, to release software to a range of users (experimental, testing) , and if nobody complains for a time, assume that it's working. This method has dubious value to the quality of actual testing going on, and while not time efficient it is definitely cost efficient. That is as long as time is not related to money.

The Development Model and Testing

The current development model is based in a few layers;

- First the Original Developer
  - codes
  - smoke tests
  - (unit tests)
  - (acceptance tests)
- Then the code reaches "other developers" (jhbuild users)
  - smoke tests
  - functional acceptance tests ("This feature doesn't work")
  - integration tests (limited to compiling and launching)
  - (unit tests)
- Then a release is rolled out, and Distributions take over
  - smoke tests
  - integration tests
  - Function tests (Gedit can't edit files!)
  - (Functional Acceptance Tests)
- The distributions then roll out to beta users
  - Function Tests
  - Acceptance Tests
  - Integration tests
- And then it reaches users

The Proposal

Without changing the world, building CI systems and forcing a culture of testing and reverts on the world, there is an somewhat easy way of changing the current state of things.

My suggestion is to write test cases for System Testing and to give our Distributors a coherent set of tests for System Testing as well.

The tests do not have to be all encompassing, but they should target the "core applications" to make sure that we retain functionality and do not regress there. Currently, there are several such test-cases both from Ubuntu, SUSE and Fedora, and organising this effort upstream appears to make sense.

This could later be coordinated with Live Images for a faster feedback loop between "Code hitting tree" and "user testing it", and would also give a basic set of functionality that should not be broken before a release of the software.

Furthermore, I suggest having these kinds of tests for all features announced in Release Notes, as those features are seen as "promises" to the community.

Other gains

Other than the immediate gains, this would give us an easier way of getting interested parties involved. By bridging the step between "just use jhbuild" and "I want to try something" we can engage more users in a smoother way.

Having this set of tests would be a way for us to indicate the features that we consider "core" to the heart, and where we have an assumption on the distributions that they should not break without very good reason.

It may also serve as a check on our own development model that we do not accidentally break work flow and user experiences during redesigns.

Coverage

We do not need to aim for complete coverage of the tests, doing so is unnecessary and will complicate things. It is better to have a small amount of tests that we can maintain, than to reach too far and become unsupported.

85% test coverage is a noble goal in Unit Testing, but is impossibly high for Acceptance Testing and System Testing.

Implementation Details

At the beginning - it's better to keep things simple, working from the Wiki and organising test cases. Based on feedback and interest from various parts, we can then branch out and maybe formalise and systematise things further.

Why not?

Why not automated? Isn't automation better?

Not in this case. Mostly for many cases we would end up spending more time running and maintaining test suites than here. Tests suites are also not proof that the software is sane and doing something useful.

This would also allow us to use and involve our large community.