Large Free Software Projects and Bugzilla: Lessons from GNOME Project QA

Luis Villa
Ximian, Inc.
Boston, MA
luis@ximian.com, http://tieguy.org/

Abstract

The GNOME project transitioned, during the GNOME 2.0 development and release cycle, from a fairly typical, mostly anarchic free software development model to a more disciplined, release driven development model. The educational portion of this paper will focus on two key components that made the GNOME QA model successful: developer/QA interaction and QA process. Falling into the first category, it will discuss why GNOME developers bought in to the process, how Bugzilla was made easier for them and for GNOME as a whole, and why they still believe in the process despite having been under Bugzilla's lash for more than a year. Falling into the second, some nuts and bolts: how the bugmasters and bug volunteers fit into the development process, how we coordinate, and how we triage and organize. Finally, the paper will discuss how these lessons might apply to other large projects such as the kernel and Xfree86.

1 Introduction

During the GNOME 2.0 development and release cycle in 2002, the GNOME project grew from a fairly typical, fairly anarchic free software development model to a more disciplined, release driven development model. A key component of this transition was the move towards organized use of Bugzilla as the central repository for quality assurance and patch tracking. This was not a process without problems- hackers resisted, QA volunteers did too little, or too much, we learned things we need to know too late, or over-engineered the process too early. Despite the problems, though, GNOME learned a great deal and as a result, GNOME's latest releases have been more stable and reliable than ever before (even if far from perfect yet :)

The purpose of this paper isn't to teach someone to do QA, or to impress upon the reader the need for good QA- other tomes have been written on each of those subjects. Instead, it will focus on QA in a Free Software context- how it works in GNOME, what needed to be done to make it work both for hackers and for QA volunteers, and what lessons can be learned from that experience. To explain these things, I'll focus on three main sections. The first will be a very brief history of GNOME's transition to a more Bugzilla-centric development style, in order to provide some background for the rest of the paper. The second part will focus on the lessons learned from this transition. If a reader needs to learn how to manage QA and Bugzilla for a large Free Software project (either as bugmaster or as a developer), this section should serve as a fairly concise guide to GNOME's free software QA best practices- explaining how developers and QA should work together, what processes make for good free software QA, and how a free software project can build a QA community that works. Finally, in the third section, the paper will attempt to discuss how GNOME's lessons might be applied to the usage of Bugzilla in other projects.

2 Background

Before going further, I'll offer a brief bit of background on the GNOME project and how it came to be a project where QA was an integrated part of the Free Software development process.

2.1 The GNOME Code Base

GNOME is not the kernel or X, but it is on roughly the same order of magnitude in terms of code and complexity. And it is continuing to grow as we integrate the functionality expected by modern desktop users. Desktop is, of course, a very vague term, but at the current time, GNOME includes a file manager, a menu system, utilities, games, some media tools, and other things you'd expect from the modern proprietary OSes. Of course, there is also a development infrastructure as well, including accessibility for the disabled, internationalization, advanced multimedia, documentation, and support for many standards.

To provide all of this, GNOME is a whole lot of code. A very basic build is more than 60 modules, each with its own build and dependencies. Wheeler's sloccount in a 'basic' build (no web browser and no multimedia infrastructure) shows 8,000 .c or .h files and roughly 2.0 million lines of code.[?]. When counting word processing, web browsing, spreadsheets, media handling, and e-mail and calendaring (all of which are provided by fairly robust, complex GTK/GNOME apps) the total grows to roughly 4.2 million lines of code.

2.1.1 The GNOME User Base

To a free software hacker, of course, users are not just users- they are potential volunteers. The 'modern desktop users' GNOME developers like and/or hate to talk about are actually quite numerous, which means a large base of people who generate bug reports (especially stack traces) and who also can be possibly persuaded to do QA work. For context, Ximian GNOME 1.4 had a million and a half installations. By late in the 2.0 cycle daily rpm builds of CVS HEAD drew thousands of downloads a day, and each tarball pre-release of GNOME 2.0 was downloaded and built by tens of thousands of testers. So, even during the relatively unstable runup to 2.0, thousands of users were pounding gnome's pre-releases on a daily basis, and many of those became willing helpers in the QA process- over 1,500 people submitted bugs during the 2.0 release cycle, and several thousand more crashes were submitted anonymously. This type of QA is difficult for anything but the largest proprietary software companies to match, and has been invaluable to GNOME.

2.1.2 QA in GNOME 2.0

As can be imagined, this much code, with this many users, trying to excersize a lot of functionality, while developers seek to make things more functional, usable and accessible, generates a lot of crash data and bug reports. Between January 2002 and the release of GNOME 2.0 on June 26th, 2002, slightly over 10,000 bugs were tagged as 2.0 bugs and an additional 13,000 non-2.0 bugs were filed. The QA team triaged or closed over 17,000 of those. Over 5,000 more were eventually marked as fixed, including over 1,000 deemed high priority by the QA team. For a project with a fairly small active core development team, these were huge numbers. It's only because of volunteer help in identifying and triaging bugs that dealing with this at all was possible. During the 2.0 cycle, regular 'bug days'- 12 hour long meetings of new volunteers- and a mailing list helped coordinate and recruit volunteers.

3 So what did we learn?

None of this was a pretty or clean process; lots of mistakes were made and not quite as many lessons learned. But we did learn a number of general rules for volunteer driven, high-volume bug handling. The gist of these lessons can be summarized simply- QA volunteers must work with their community to find, identify, and track the most important bugs. But the details are more complex and will, I hope, make this paper worth reading.

3.1 QA and Hackers

While ESR may have his critics, he was undoubtedly right in observing that we are all in this to scratch our itches, whatever those itches may be. Free software QA is a slightly odd bird in this light- QA volunteers are in it to scratch the itch of higher quality software, but they can't do it themselves. That means paying a lot more attention to the needs of others than may be typical for free software. Following are some of the GNOME team's lessons learned.

3.1.1 Rule 1: free software QA must support the needs and desires of developers in order to succeed

This seems fairly obvious, but it is also fairly easy to forget or ignore. Free software begins and ends with developers who are having fun. There may be others involved for reasons other than 'fun', but if QA's sole purpose is to whine about what QA thinks are the important flaws, volunteers will leave, and leave quickly. QA must think first and foremost not about their own goals, but about the goals of developers.

To put it another way: developers think they can do their thing without QA (which, in free software, they mostly can) and QA absolutely cannot do its job (which is getting bugs fixed, not just finding bugs) without developers. If QA forgets this in proprietary software, developers have to suck it up. If QA forgets this in free software, developers will ignore them, or worse, walk away from the project.

This is not to say that QA must be silent servants of hackers, never giving feedback or their own input. QA volunteers can be and should be trusted individuals whose advice is valued. But that will happen more quickly and more easily if the goal of supporting and aiding hackers is always first and foremost.

3.1.2 Rule 2: QA needs guidance from maintainers

In order for QA volunteers to serve the needs of maintainers and developers in general, maintainers and developers must clearly communicate their priorities. This falls out pretty cleanly from rule 0: if a project doesn't know where the project is going, or what the project's developers want, then it is going to be very hard for QA to help reach those goals. This also means that when a project is conflicted, QA teams may not be of as much utility as a project expects.

'What a project wants', from a QA perspective, is usually pretty obvious in GNOME- stability, stability, stability. If a program can be crashed, or a button doesn't work, everyone typically agrees that this should raise a red flag. But past that, things often get murkier- some maintainers may, for example, care deeply about code quality, while others may be deeply involved with fixing usability issues. And how does one weigh a difficult to reproduce crasher against, say, a build problem on Solaris? That's not typically an easy or fun call to make; it's nearly an impossible one to make correctly unless a QA volunteer has guidance from the developer.

In proprietary QA, these answers are usually pretty easy- compare against a design doc and go. In free software QA, where design docs are often lacking in details if they exist at all, developers must do the best they can to communicate to QA what exactly the priorities are so that when the QA team finds a problem they can classify it correctly.

3.1.3 Rule 3: QA must persuade hackers they are useful and intelligent

When I came on board the Evolution team, the universal response was 'oh, someone is going to mark duplicates for us, that's nice.' When I left, I was very flattered to know that the team was worried about a lot more than duplicates. The difference between coming and going was not just that I was effective, but that the very first thing I did was work very hard to learn the lay of the land in Evolution. Instead of reading one bug and deciding 'this is bad', I read nearly two thousand bugs before doing anything more than rudimentary marking of duplicates in the bug database. This is an extreme example, of course, but it is the direction Free Software QA volunteers should lean if they can.

In contrast, some first-time GNOME volunteers have dropped in, read one or two bugs, and decided 'oh, this bug is hugely important', and tried to mark it as such. Worse, some will try to guess at the source of problems in code they've never looked at, or (this is inevitably the most irritating to developers) they'll declare that something 'must be easy to fix'. Invariably, this leads to irritation from developers who have seen a lot more issues and have, unsurprisingly, a much better grasp of their own code. The best way to avoid this is to work hard to always make the right call, especially when first working on a project. There aren't the obvious checks of functionality and code review that typically establish trust between hackers- so very sound and conservative judgement has to substitute at first.

3.1.4 Rule 4: Bugzilla cannot be the end-all and be-all of communication between hackers and QA

Bugzilla is a wonderful tool, that allows for great communication and incredibly flexible ways to sort, parse, and otherwise mangle bugs. But it doesn't speak to mailing lists, and it can't selectively poke hackers about important issues. QA volunteers must actively seek out other, non-Bugzilla forms of communication- mailing lists and IRC, primarily, but also web pages and other forums. Use these channels to draw attention to QA and to QA processes- most important new or outstanding bugs, important recent fixes, new features or reports in Bugzilla, or even simple 'this many bugs were opened and this many closed last week.' By doing this, a QA team can establish an identity as a 'regular' part of the development process even amongst developers who aren't familiar or comfortable with Bugzilla.

This was a lesson learned the hard way in GNOME. During the 2.0 cycle, the bug squad assembled and emailed regular Friday status reports to GNOME's main development list. It was well recieved by hackers, but like many things in free software, it wasn't completely appreciated until (during the 2.2 cycle) it was gone, done away with by lack of attention on the part of the bug squad and the mistaken belief that it wasn't very useful to developers. Developers let us know, and as a result we'll try to bring

3.2 Triage

Triage is a word that has been thrown around in this paper a fair amount- before going further it may be useful to define it. In medicine, battlefield triage is the process of separating the very badly wounded from those who are lightly wounded and those who are so wounded that they will die regardless of treatment. In a free software context, it's the process of separating and identifying bugs that are most severe and/or most useful to developers out from the inevitable mountain of bugs that will come in for any popular project. Specifically, in GNOME, we triage by setting 'priority', 'severity', and 'milestone' fields in Bugzilla. Like communication, GNOME has learned a fair amount about this in the past year.

3.2.1 Rule 5: bugs need to be triaged, not just tracked

When I came into GNOME and Evolution, both projects knew that having a Bugzilla was a good thing. So, they dutifully entered bugs in their bug tracking systems- they tracked bugs. But neither project had useful definitions of severity and priority- they couldn't or didn't triage their bugs. So what they had, when they needed to know what came next, was a large list of bugs in basically random order. Not surprisingly, that wasn't very helpful and so bugs ended up getting entered in to Bugzilla and never read again. Developers ended up maintaining lists outside of Bugzilla to help them figure out what to do next- a silly duplication inefficiency in projects that can't really afford much inefficiency.

If this kind of thing is happening, it indicates that bugzilla is not being used properly. The solution is to carefully define priorities, severities, and milestones, and use them religiously, by looking at every bug and making at least an attempt to judge how bad it is and when it should be fixed by. When it comes down to release time, having consistently marked bugs with these priorities means that it will be much easier to say 'these things must be fixed, these we fix if we have time, these we pretend just don't exist.' And that will leave you with much better software.

3.3 Rule 6: triage must be tied to release goals

This is a whole lot like Rules 0 and 1, so I'll be brief. It's worth repeating, though- triage is basically the art of determining what is important, and if QA and hackers frequently disagree on what is important, QA will get ignored. This greatly reduces the space for personal freedom in QA- several volunteers have come into GNOME, picked up on a pet theme and marked those bugs up, and I've spent a great deal of time apologizing for them. Bugzilla cannot be allowed to become a soapbox, for anything except the goals maintainers have already agreed to. If there is dissent on those goals, take it to the lists- Bugzilla is not a good forum for setting or arguing goals.

(In proprietary software, this is easy- 'project goals' are in a tightly defined project spec that must be followed. Bug volunteers, especially those coming from a proprietary background, must remember that this just isn't possible in Free Software.)

3.3.1 Rule 7: triage new bugs agressively, or Bugzilla will quickly terrify maintainers

The initial temptation of almost all the QA volunteers I've dealt with is to assume that a bug they've just read is extremely important, and should be prioritized to reflect that. In some ways, this is true- we do see a lot of very ugly bugs, that in an ideal world given infinite resources and time would be high priority. But we live in a world of volunteers and spare time, so marking bugs as more important than others should be done only carefully and conservatively.

Most free software hackers work on their projects in blocks of very short periods of time. That means that if Bugzilla is their TODO list, the smaller and the more sorted it is, the more beautiful. In practice, we've found that it means that once maintainers trust their QA, they tend to only look at high priority bugs. This can be scary- it puts a lot of power in the hands of QA, and messing up by deciding that a bug is not important enough for a maintainer to look is seemingly very bad. This is utterly true in proprietary QA- if a QA guy screws up and punts something that he shouldn't, there may not be much of a system of checks and balances to catch the error. Free Software QA saves us from such a fate- punt a bug or mark it low priority, and if it is important, ten other people will file it or add comments. The massive volume of bugs we get is a constant check.

For example, in GNOME, we regularly see crashes that a maintainer or QA volunteer (or often the original reporter) decides is completely impossible to reproduce. We knock them down in importance or close them in that case. Often, they actually are impossible to reproduce- build problem, transient issue that got fixed the next day, or other such. But in some cases, after everyone has thrown up their hands, you'll continue to see reports of the crash. The 'mistake' of triaging or punting the original crash can then be revisited- thanks to the volume of bugs we recieve, we've gotten ample confirmation that maybe it wasn't such a bogon after all.

This isn't perfect, of course- in Evolution, for example, we get relatively few bugs on first-time installation, so a single punt on an installation issue may obscure much deeper and more important issues that won't be filed again for some time. But, unfortunately, it's something that frequently must be done- the alternative is often for maintainers to query Bugzilla and face massive lists that are quickly overwhelming. QA can and should serve as a buffer for that if necessary.

3.3.2 Rule 7: closing old bugs, even completely unread, is unpleasant but OK

GNOME's QA was publicly flamed several months ago by someone (we'll call them 'james w. z.') for mass-closing old GNOME bugs without substantially reviewing them. This was an unfortunate thing that we hate to do, but it was justified. In the typical free software cycle, a project starts off too unstable and with too few users to get many bug reports. After the project builds and grows, you still have all the old bugs from the early period, and an increasing number of users and bug reporters, many of whom are filing bugs you can't possibly have time to fix or even sometimes look at before your next rewrite and release.

Faced with an escalating number of bugs, a volunteer-driven project that can't easily bring in more resources has two options: mass close old bugs with an 'if you still see this in our latest release, please reopen the bug', or let the DB grow so large that it is unusable for hackers and QA volunteers alike. From these two, the choice is obvious if unpleasant. Furthermore, as 'james' reminded us, this isn't something that is easy for bug filers to understand. But when doing it, remind yourself: if it is still a bug, someone will file it again.

3.3.3 Rule 8: triage rules can't be just in one person's head

As already mentioned, the first step I took when moving in to GNOME was to revise and rewrite GNOME's definitions for priorities. Previously, they'd been fairly broad and inspecific. The new priorities gave specific examples, and tried to group problems into specific classes as well. This was an important first step for sane triage across Bugzilla. But it was not enough- nearly all judgment calls by volunteers ended up coming back to me for validation, since the definitions did not include a lot of my experience and judgement- just examples and definitions. So, I've started (and the QA volunteers have rewritten and completed) a GNOME triage guide [http://developer.gnome.org/projects/bugsquad/triage/]. This document attempts to put a lot of collective wisdom down onto paper, and makes it easier for new volunteers to come in and get started, and for old volunteers and developers to understand more precisely what should be going on.

This will hold true for any project without strong guidelines, I believe- either a large group of volunteers will inconsistently apply their own judgments (confusing developers) or the project will become overly dependent on one person, which will eventually again lead to inconsistency as the mass of bugs becomes too much for that one person to handle. Again, this was a lesson learned by GNOME only after 2.0- during the 2.0 cycle, much of the triage wisdom stayed in my head and when I had less time (during the 2.2 cycle) the process grew a bit creaky, because triage often blocked on my availability to answer judgment calls.

3.4 Some Miscellaneous (But Important) Observations on Free Software QA

There are a few other important lessons GNOME has learned that aren't rules, per se, but which everyone trying to do Free Software QA should always keep in mind.

3.4.1 Observation 1: volunteers and hackers are expensive, and bugs are cheap

You could also think of this as 'volunteers are scarce, bug filers are like locusts.' This has a number of implications for Free Software QA- many of the rules I've previously cited are almost the direct results of this observation. Many others I haven't cited also fall out of it. If you keep this simple observation (almost more a law than a rule) in mind, you'll find the others with time.

3.4.2 Observation 2: triage is an imperfect art

Despite the immediately previous suggestions about how to make triage consisten, it must be understood that triage is an imperfect art, where a certain amount of inconsistency is inevitable.

As already mentioned, the best way to triage is to read a lot of bugs first, to gain an appreciation for what types of bugs a project is seeing and how severe they are. But even after having read 20,000 or so bugs in the past year, over four projects, drawing the line even between seemingly simple things like 'is this an enhancement or a bug' is a frequent borderline judgement call for me.

Everyone involved in the QA process- bug reporters, bug fixers, and bug triagers (both casual and regular) must learn to accept this and work with it. The important lesson here is that volunteers should not be held to an impossible standard- both volunteers and developers must understand that differences of opinion will happen and aren't the end of the world. There will be thousands more bugs to work with if one gets screwed up. :)

3.4.3 Observation 3: QA is winning when people are interested in the process, not just the results

So how can one know when QA is starting to win? At what point can a QA volunteer sit back and say 'the hard part is done, now all I have to do is read bugs?' I'd suggest that one important metric is noting the point when the standard response by developers to bug reports is 'put it in Bugzilla.' GNOME moved very slowly in this direction, but that's now pretty much the standard response on mailing lists when a bug is reported to a list- 'take it to Bugzilla.' There are other parts of the process as well- bug days, noting bug numbers in CVS commits or code comments, and an overall commitment by developers to working with QA volunteers.

4 And these rules apply to other large projects how?

4.1 XFree

A few months back, XFree had a discussion on their mailing list about use of Bugzilla to track XFree86 bugs. The response was... underwhelming. Why? The main fears were pretty straightforward: 'will we get lots of useless bug email?', 'will people try to control what we do?', and of course 'what benefit do we get?'

The answers to these questions aren't always obvious to a project just embarking on doing serious Free Software QA for the first time. Being more public can definitely open maintainers up to more mail. Obviously, this can happen- as I discussed, it's even possible that less buggy software will get more bug reports. That's not truly a requirement- even if Bugzilla is used only to triage and track bugs that come in through other forums (say, open only to developers, and used to track issues reported to a mailing list) it can still be of great use to a project, assuming that other rules I laid out about supporting developer goals and defining the triage process well are followed. GNOME actually allows anonymous bug submission, the opposite end of the spectrum, and while this is far from perfect, it has helped us make huge leaps and bounds in terms of stability by encouraging stack-trace submission.

Concerns about 'control' were equally unfounded- even borderline paranoiac. Xfree, like all other Free Software projects, is controlled by hackers and hackers alone. If a hacker decides that QA volunteers aren't to be trusted, or simply disagrees with triage decisions, they can ignore them and move on. The burden is on those running the QA process to prove that their bugs are valid and useful. I've given some suggestions on how this can happen already.

Finally, the most obvious and at the same time most difficult question- 'what benefit do we get?' I got into Linux because I'd heard about, heard it didn't crash, and one night Outlook crashed 10 times, while I was trying to write a single email to a professor. So for me, more stable software is an obvious benefit of working in QA. That is, admittedly, not for everyone. Answering this question really requires some introspection on the part of hackers and maintainers- if you want to make software that is good for your users (virtually no matter how you define good), then your project wants a QA process and wants Bugzilla. If you are in free software purely because you want to write cool hacks, or because in free software, no one can tell you what to do, Bugzilla may not be for your project. But that's an answer only you can answer. Frankly, on reading the XFree lists, it was not altogether clear that many of the XFree hackers were particularly concerned about their users. If that is the case (and that is most definitely their prerogative as authors of the code) then perhaps Bugzilla is not for them. No matter how hard they try, it would be hard for QA volunteers to support the pursuit of power or cool hacks.

4.2 Kernel

Reading kernel traffic[FIXME: cite issue] I was very pleased to see that Andrew Morton had proposed not just a list of bugs, but actually defined what he felt should be the standard for 'when should we go to 2.6.0?' I'm a long time k-t reader, and this was the first time I'd seen something of the sort. Defining and agreeing on that is part of my Rule 0- QA has to support development, and developers have to tell QA what they want. The list had even been split into rudimentary 'can't ship without fixing' and 'it would be nice' lists- a big first step towards solid triage.

It was sort of disappointing, though, to read through the details- the vast majority of issues had no bugs associated with them, and squirelled down at the bottom was 'and there are several hundred open Bugzilla bugs.' This was the kind of opportunity kernel bug people should have seized on (and possibly have since this paper has been written, of course.) Bugzilla is perfectly designed to track these kinds of issues and their progress. Some intrepid volunteer could easily have volunteered to enter every issue into Bugzilla and assign it a high priority and assign it to the owner of the issue. From there, patches could have been attached and tracked, punting it from one list to another would have been as simple as changing a single field, outsiders could easily discover what bugs had and hadn't been fixed already, and a host of other things. Instead of ongoing IRC status meetings where many things were reported fixed or irrelevant, a simple query could have reported a list before the meeting that could be updated by all participants in parallel if need be. (And of course, no more diffs to show what had changed- again, simple, dynamic query to show what has changed over any period of time.)

Similarly, the 'several hundred open Bugzilla bugs' was a great invitation for someone to work with Bugzilla, trawl them (it only takes a weekend, at worst, to read a couple hundred bugs, once you've got the knack) and start making preliminary suggestions to maintainers aobut important bugs that were in Bugzilla but not on the list. Remember rule two- persuade the hackers you are useful. Filling in the blanks on information they knew must be there but didn't have time to find themselves is a great step towards that, and reading the (currently small) open bugs to get perspective would have been a great start for those looking to help out and do effective triage.

Pre-release is the best time for QA and Bugzilla- priorities are typically clear cut, hackers are most pressed for time and so most appreciative of the help, and hackers are the most motivated to work on bug-fixing instead of pie-in-the-sky features. Hopefully, someone involved in the kernel community will find this general advice useful and can take advantage of this relatively rare time in the kernel cycle.

5 Conclusion

Free Software QA is a slightly different beast, playing with different sets of data inputs and different sets of motivations than a typical QA process. As a result, making QA central to the release process is not easy for any Free Software project, and it's even harder to stay with it once it is successful, since success breeds difficulty. But it can be done if communication, motivation and technique are all brought in line with each other. GNOME did, and benefited immensely from it. It is my hope that other large projects will be able to learn from our lead.

6 Acknowledgments

Thanks to Ximian and Sun, for allowing me to work so extensively with the GNOME community.

Thanks to the Bugsquad and all the volunteers who preceded it, first for doing so much work for their own communities, and second for keeping me sane while I worked on Evolution and GNOME. And thirdly for suggesting the title of my talk.

Thanks to all those who proceeded me, at Mozilla and GNOME, for giving me something to work with- tools, skills, and data.

Thanks to ed on gimpnet, for helping me fight through the structure of this paper.

7 Availability

This paper and slides from the associated presentation will be available from

`http://tieguy.org/talks/`

References

[Wheeler] From David Wheeler's SLOCCount- http://www.dwheeler.com/sloccount/

03