Communicating your project: honest marketing for free software projects

Communicating your project is an essential step for **getting the users you want**. *If you're **pressed for time** and want the really short form, just jump to the questionnaire.*

> You have an awesome project, but you see people reach for inferior tools? There are people using your project, but you can't reach the ones you care about? Read on for a way to ensure that your communication doesn't ruin your prospects but instead **helps your project to shine**.

In this article I summarize my experience from working on several different projects including KDE (*where I learned the basics of PR - yay, sebas!*), the Hurd (*where I could really make a difference by improving the frontpage and writing the Month of the Hurd*), Mercurial (*where I practiced minimally invasive PR*) and 1d6 (*my own free RPG where I see how much harder it is to do PR, if the project to communicate is your own*).

Since voicing the claim that marketing is important often gets you into discussions with people who hate marketing of any kind, I added an appendix which illustrates with an actual example what happens if you don't do any PR - and what happens if you do PR of the wrong kind.

> They just want to know what you're making and why you're making it and who's it for.
>
> — Michael Douse
> Director of Publishing, Larian (Baldur's Gate 3)
> 2024-04-25 by PC Gamer, 35:46

Contents

1 What is good marketing? 2

2 How to communicate your project? 3
2.1 Who are our Target Groups? . . . 3
2.2 What could they ask? . . . 5
2.3 Whose wishes can we fullfill? . . . 5
2.4 Provide those answers! . . . 6
2.5 Further points . . . 7

3 bab-com q: Project Communication Questionaire 8
3.1 For whom are we already **useful or interesting**? Name them . . . 8
3.2 **Whom do we want** as users on the long run? Name them . . . 8
3.3 What could they **ask**? What are **their needs**? Write questions . . . 8
3.4 **Answer** their questions . . . 8
3.5 Whose needs can we already fulfill? Are we the **best choice**? . . . 8

4 Note: The mission statement and the slogan 9
4.1 Screenshots and quotes . . . 9

5 Summary 10

6 Appendix: Why communicating your project? 10

1 What is good marketing?

Before we jump directly to the guide, there is an important term to define: Good marketing. That is the kind of marketing, we want to do. The definition I use here is:

> **Good marketing** *ensures that the* ***people to whom a project would be useful*** *learn about the project.*

and

> **Good marketing** *starts with the* ***existing strengths*** *of a project and finds people to whom these strengths are useful.*

Thus it does not try to interfere with the greater plan of the project, though it might identify points where a little effort can make the project much more interesting. Instead it finds people to whom the project can be useful and ensures that they know it.

> *Be fair to competitors, be honest to users, put the project goals before generic marketing considerations.*

As such, good marketing is an interface between the project and its (potential) users.

2 How to communicate your project?

This guide depends on one condition: Your project already has at least one area in which it excels over other projects. If that isn't the case, please start by making your project more useful to some people.

> To make this text easier to follow, I give examples from the latest project where I did this analysis: GNU Guile: The GNU Ubiquitous Intelligent Language for Extensions. Guile provides a nice example, because its mission is clearly established in its name and it has lots of backing, but up until our discussion actually had a wikipedia-page which was unappealing to the point of being hostile against Guile itself.

The basic way for communicating your project to its potential users always follows the same steps:

- identify the **target groups** of the project
- find their **questions**
- **answer** their questions
- **choose** the strong answers
- **provide** those answers in your communication

2.1 Who are our Target Groups?

To improve the communication of our project, we first **identify our *target groups***.

We begin by asking ourselves, who would profit from our project:

- What can we do well and how do we compare to others?
- To whom would we already be useful or interesting if people knew our strengths?
- To whom are we already the best option?

Try to find 3 groups of people and give them names which identify them. Those are the people we must reach to grow on the short term.

In the next step, we ask ourselves, whom we want or need as users to fullfill our mission (our long-term goal):

- Where do we want to get? What is our goal? (do we have a mission statement?)
- Whom do we need to get there?

- Whom do we want as users? Those shape us as they take part in the development - either as users or as fellow developers.

Try to find 2 groups of people and give them names which identify them. Those are the people we must reach to achieve our longterm goal. If while writing this down you find that one of the already identified groups which we could reach would actually detract us from our goal, mark them. If they aren't direly needed, we would do best to avoid targeting them in our communication, because they will hinder us in our longterm progress: They could become a liability which we cannot get rid of again.

Now we have at least 5 *target groups*: Those are the people who should know about our project, either because they would benefit from it for pursuing their goals, or because we need to reach them to achieve our own goals.

2.1.1 Example: Target Groups for Guile

GNU Guile is called The GNU Ubiquitous Intelligent Language for Extensions. So its mission is clear: Guile wants to become the de-facto standard language for extending programs - at least within the GNU project.

2.1.2 For whom are we already useful or interesting? Name them as Target-Groups.

1. Schemer: Wants to see what GNU Scheme can do.
2. Extender: GNU enthusiast wants to extend an existing program with scripting
3. Learner: Free Software enthusiast thinks about using Guile to learn programming
4. Project-Starter: Experienced Programmer wants to start a new project.
5. 1337: Programmer wants the coolness-factor.
6. Emacser: Emacs users want to program in Guile using Emacs.

2.1.3 Whom do we want as users on the long run? Name them as Target-Groups.

7. GNU-folk: All GNU developers.

2.2 What could they ask?

We now need to find out which kind of information our target groups actually need or search.

This part requires thinking ourselves into the role of each of the target groups. For each of the target groups, ask yourself:

> *What would you want to know, if you were to read about our project?*

As result of this step, we have a set of answers. Judge them on their strengths: Would these answers make you want to invest time to test our project? If not, can we find a better answer?

2.2.1 Example: Questions for the Target-Groups of Guile

1. Schemer: What can guile do better than other Schemes?
2. Extender: What does Guile offer my program? Why Guile and not Python/Lua?
3. Learner: How easy and how powerful is Guile Scheme? Why Guile and not Python?
4. Starter: What's the advantage of starting my advanced project with guile?
5. 1337: Why is guile cool?
6. Emacser: What does Guile offer for Emacs?
7. GNU-folk: What does Guile offer my (favorite) program? (Being a GNU package is a distinct advantage, so there is less competition by non-GNU languages)

2.3 Whose wishes can we fullfill?

If our answers for a given group are not yet strong enough, we cannot communicate our project convincingly to them. In that case it is best to postpone reaching out to that group, otherwise they could get a lasting weak image of our project which would make it harder to reach them when we have stronger answers at some point in the future.

Remove all groups whose wishes we cannot yet fullfill, or for whom we do not see ourselves as the best choice.

2.3.1 Example: Chosen Target-Groups

1. Schemer: Guile is a solid implementation of Scheme. For a comparison, see An opinionated Guide to Scheme implementations.
2. Extender: The guile manual offers a nicely detailed guide for extending a program with Guile. We're a bit weak on the examples and existing extensions, though, especially on non-GNU-plattforms.
3. Learner: There are tutorials for learning to program in Guile (Naming & Logic and Scheme primer), in addition to some about Scheme in general - and even one for understanding Scheme from the view of a Python-user. But our project resources cannot yet support people who cannot program at all well enough, so we have to restrict ourselves to programmers who want to learn a new language.
4. Starter: Guile has solid support for many unix-specific things, but it is not yet a complete publishing solution for any kind of project to any platform. Naming & Logic explains how to start web projects as well as arbitrary projects in environments with proper package management (mostly GNU/Linux).
5. 1337: Guile is explicitely named in the GNU Coding Standards. It doesn't get much cooler than that - at least for a certain idea of cool. We can't get the Java-1337s, but we can get the Free Software-1337s.
6. Emacser: Geiser provides solid Guile Scheme support in Emacs.
7. GNU-folk: They are either extenders or project starters or learners, but they need information on existing support for their favorite program.

2.4 Provide those answers!

Now we have answers for the target groups. When we now talk or write about our project, we should keep those target groups in mind.

You can make that arbitrarily complex, for example by trying to find out which of our target groups use which medium. But lets keep it simple:

Ensure that our website (and potentially existing wikipedia page) includes the information which matters to our target groups. Just take all the answers for all the target groups we can already reach and check whether the basic information contained in them is given on the front page of our website.

And if not, find ways to add it.

As next steps, we can make sure that the questions we found for the target groups not only get answered, but directly lead the target groups to actions: For example to start using our project.

2.4.1 Example: The new Wikipedia-Page of Guile

For Guile, we used this analysis to fix the Wikipedia-Page. The old-version mainly talked about history and weaknesses (to the point of sounding hostile towards Guile), and aside from the latest release number, it was horribly outdated. And it did not provide the information our target groups required.

The current Wikipedia-Page of GNU Guile works much better - for the project as well as for the readers of the page. Just compare them directly and you'll see quite a difference. But aside from sounding nicer, the new site also addresses the questions of our target groups. To check that, we now ask: Did we include information for all the potential user-groups?

1. Schemers: Yepp (it's scheme and there's a section on Guile Scheme
2. Extenders: Yepp (libguile)
3. Learners: Not yet. We might need a syntax-section with some examples. But wikipedians do not like Howto-Like sections. Also the interpreter should get a notice.
4. Project-Starters: Partly in the "core idea"-part in the section Guile Scheme. It might need one more paragraph showing advantages of Guile which make it especially suited for that.
5. 1337s: It is the preferred extension system for the GNU Project. If you're not that kind of 1337: The Macro-System is hygienic (no surprising side-effects).
6. Emacs users: They got their own section.
7. GNU-folk: Not yet: No section on Guile support in existing GNU Projects.

So there you go: Not perfect, but most of the groups are covered. And this also ensures that the Wikipedia-page is more interesting to its readers: A clear win-win.

2.5 Further points

Additional points which we should keep in mind:

- On the website, do all of our target groups quickly find their way to *advanced information* about their questions? This is essential to keep the ones interested who aren't completely taken by the short answers.
- What is a common negative misconception about our project? We need to ensure that we do not write anything which strengthens this misconception. Is there an existing strength, which we can show to counter the negative misconception?
- Where do we want to go? Do we have a mission statement?

3 bab-com q: Project Communication Questionaire

3.1 For whom are we already useful or interesting? Name them

1.
2.
3.

3.2 Whom do we want as users on the long run? Name them

4.
5.

3.3 What could they ask? What are their needs? Write questions

1.
2.
3.
4.
5.

3.4 Answer their questions

1.
2.
3.
4.
5.

3.5 Whose needs can we already fulfill? Are we the best choice?

1.
2.
3.
4.

Ensure that our communication **includes the answers** to the questions of those groups (i.e. website, wikipedia page, talks, ...), at least for the groups who are likely to use the medium on which we communicate!

> *Use bab-com to avoid bad-com :-) - yes, I know this phrase is horrible, but it is catchy and that fits this article: you need catchy things.*

4 Note: The mission statement and the slogan

The questionaire helps you to get a good project description. But first you need to get people interested enough to actually read that. This is where mission statement and slogan come in.

The **mission statement** is a short paragraph in which a project defines its goal. It's your 10-30 seconds elevator pitch.

A good example is:

> *Our mission is to create a general-purpose kernel suitable for the GNU operating system, which is viable for everyday use, and gives users and programs as much control over their computing environment as possible.* → GNU Hurd mission explained

Another example again comes from Guile:

> *Guile was conceived by the GNU Project following the fantastic success of Emacs Lisp as an extension language within Emacs. Just as Emacs Lisp allowed complete and unanticipated applications to be written within the Emacs environment, the idea was that Guile should do the same for other GNU Project applications. This remains true today.* → Guile and the GNU project

Closely tied to the mission statement is the **slogan**: A catch-phrase which helps anchoring the gist of your project in your readers mind. Guile does not have that, yet, but judging from its strengths, the following could work quite well for Guile 2.0 - though it falls short of Guile in general:

> *GNU Guile scripting: Use Guile Scheme, reuse anything.*

4.1 Screenshots and quotes

You need screenshots. Something to look at. Even if its just a code example with source highlighting. And quotes: people recommending you. For the target groups.

5 Summary

We saw why it is essential to communicate the project to the outside, and we discussed a simple structure to check whether our way of communication actually fits our projects strengths and goals.

Finding the communication strategy actually boils down to 3 steps:

- Focus on those who would profit from our project or whom we need.
- Check what they need to know.
- Answer that.

Also a clear mission statement, slogan and project description help to make the project more tangible for readers. In this context, good marketing means to ensure that the right people learn about the real strengths of the project.

With that I'll conclude this guide. Have fun and happy hacking!
— Arne Babenhauserheide

---

6 Appendix: Why communicating your project?

In free software we often think that quality is a guarantee for success. But in just the 10 years I've been using free software nowadays, I saw my share of technically great projects succumb to inferior projects which simply reached more people and used that to build a dynamic which greatly outpaced the technically better product.

One example for that are pkgcore and paludis. When portage, the package manager of Gentoo, grew too slow because it did ever more extensive tests, two teams set out to build a replacement.

One of the teams decided that the fault of the low performance lay in Python, the language used by portage. That team built a package manager in C++ and had `--wonderfully-long-command-options` without shortcuts (have fun typing), and you actually had to run it twice: Once to see what would get installed and then again to actually install it (while portage had had an `--ask` option for ages, with `-a` as shortcut). And it forgot all the work it had done in the previous run, so you could wait twice as long for the result. They also had wonderful latin names, and they managed the feat of being even slower than portage, despite being written in C++. So their claim that C++ would be magically faster than python was simply wrong (because they skipped analyzing the real performance bottlenecks). They called their program paludis.

Note: Nowadays paludis has a completely new commandline interface which actually supports short command options. That interface is called `cave` and looks sane.

The other team did a performance analysis and realized that the low performance actually lay with the filesystem: The portage tree, which holds the required information, contains about 30,000 ebuilds and almost 200,000 files in total, and portage accessed far more of those files than actually needed for resolving the dependencies needed to install the package. They picked python as their language - just like portage. They used almost the same commandline options as portage, except for the places where functionality differed. And they actually got an order of magnitude faster than portage - so fast that their search command often finished after less than a second, while portage took over 10 seconds. They called their program pkgcore.

Both had more exact resolution of packages and could break cyclic dependencies and so on.

So, judging from my account of the quality, which project would you expect to succeed?

I sure expected pkgcore to replace portage within a few months. But this is not what happened. And as I see it in hindsight, the difference lay purely in PR.

The paludis team with their slow and hard-to-use program went all over the Gentoo forums claiming that Python is a horrible language and that a C program will beat portage any time. On their website they repeated their attacks against python and claimed superiority at every step. And they gathered quite a few zealots. While actually being slower than portage. Eventually they rebranded paludis as just better and more correct, not faster. And they created their own distribution (exherbo) as direct rival of Gentoo. With a new, portage-incompatible package format. As if they had read the book, how *not* to be a friendly competitor.

The pkgcore team on the other hand focussed on good technology. They created the snakeoil library for high-performance python code, but they were friendly about it and actually contributed back to portage where code could be shared. But their website was out of date, often not noting the newest release and you actually had to run `pmerge --help` to see the most current commandline options (though you could simply guess them if you knew portage). And they got attacked by paludis zealots so much, that this year the main developer finally sacked the project: He told me on IRC that he had taken so much vitriol over the years that it simply wasn't worth the cost anymore.

> *Update*: About a year later someone else took over. Good code often survives the loss of its creator.

So, what can we learn from this? Technical superiority does not gain you anything, if you fail to convince people to actually use your project.

> *If you don't communicate your project, you don't get users. If you don't get users, your chances of losing motivation are orders of magnitude higher than if you get users who support you.*

And aggressive marketing works, even if you cannot actually deliver on your promises. Today paludis improved its user-interface and even has short option-names. But even to date, exherbo has much fewer packages in its repositories than Gentoo. If the number of files is any measure, the 10,000 files in their special repositories are just about 5% of the almost 200,000 files portage holds. But they managed quite well to fraction the Gentoo users - at least for some time. And their repeated pushes for new standards in the portage tree (EAPIs) created a constant pressure on pkgcore to adapt, which had the effect that nowadays pkgcore cannot install from the portage tree anymore (the search still works, though, and I still use it - I will curse mightily on the day they manage to also break that).

> *Update*: Someone else took over and now pkgcore can install again.

So aggressive marketing and doing everything in the book of unfriendly competition might have allowed the paludis devs to gather some users and destroy the momentum of pkgcore, but it did not allow them to actually become a replacement of portage within Gentoo. Their behaviour alienated far too many people for that. Aggressive and unfriendly marketing is better than no marketing, but it has severe drawbacks which you will likely want to avoid.

> *If you use overly aggressive, unfriendly or dishonest communication tactics, you get some users, but if your users know their stuff, you won't win the mindshare you need to actually make a difference.*

If on the other hand you want to see communication done right, just take a look at KDE and Gnome nowadays. They cooperate quite well, and they compete on features and by improving their project so users can take an informed choice about the project they choose. And their number of contributors keeps growing steadily.

So what do they do? Besides being technically great, it boils down to good marketing.

List of Links

draketo.de: https://www.draketo.de . . . 1
KDE: http://kde.org . . . 1
Hurd: http://hurd.gnu.org . . . 1
Mercurial: http://mercurial.selenic.com . . . 1
1d6: http://1w6.org/english/flyerbook-rules . . . 1
2024-04-25 by PC Gamer, 35:46: https://youtu.be/89j58d6Gk1I?t=2147 . . . 1
GNU Guile: The GNU Ubiquitous Intelligent Language for Extensions: http://www.gnu.org/software/guile/ . . . 3
GNU Guile is called The GNU Ubiquitous Intelligent Language for Extensions: http://www.gnu.org/software/guile/ . . . 4
An opinionated Guide to Scheme implementations: http://wingolog.org/archives/2013/01/07/an-opinionated-guide-to-scheme-implementations . . . 6
detailed guide for extending a program with Guile: http://www.gnu.org/software/guile/manual/guile.html#Programming-in-C . . . 6
Naming & Logic: https://www.draketo.de/software/programming-scheme.html . . . 6
Scheme primer: https://files.spritely.institute/papers/scheme-primer.html . . . 6
understanding Scheme from the view of a Python-user: http://phyast.pitt.edu/~micheles/scheme/ . . . 6
Naming & Logic: https://www.draketo.de/software/programming-scheme.html . . . 6
explicitely named in the GNU Coding Standards: http://www.gnu.org/prep/standards/standards.html#Source-Language . . . 6
Geiser: http://www.nongnu.org/geiser/ . . . 6
old-version: http://en.wikipedia.org/w/index.php?title=GNU_Guile&oldid=564014065 . . . 7
current Wikipedia-Page of GNU Guile: http://en.wikipedia.org/wiki/GNU_Guile . . . 7
GNU Hurd mission explained: http://www.gnu.org/software/hurd/community/weblogs/antrik/hurd-mission-statement.html . . . 9
Guile and the GNU project: http://www.gnu.org/software/guile/manual/html_node/Guile-and-the-GNU-Project.html . . . 9