Matthew Paul Thomas

Why Free Software has poor usability, and how to improve it

When I wrote the first version of this article six years ago, I called it "Why Free Software usability tends to suck". The best open source applications and operating systems are more usable now than they were then. But this is largely from slow incremental improvements, and low-level competition between projects and distributors. Major problems with the design process itself remain largely unfixed.

Many of these problems are with *volunteer* software in general, not Free Software in particular. Hobbyist proprietary programs are often hard to use for many of the same reasons. But the easiest way of getting volunteers to contribute to a program is to make it open source. And while thousands of people are now employed in developing Free Software, most of its developers are volunteers. So it's in Free Software that we see volunteer software's usability problems most often.

That gives us a clue to our first two problems.

1. **Weak incentives for usability.** Proprietary software vendors typically make money by producing software that people want to use. This is a strong incentive to make it more usable. (It doesn't always work: for example, Microsoft, Apple, and Adobe software sometimes becomes worse but remains dominant through network effects. But it works most of the time.)

   With volunteer projects, though, any incentive is much weaker. The number of users rarely makes any financial difference to developers, and with freely redistributable software, it's near-impossible to count users anyway. There are other incentives — impressing future employers, or getting your software included in a popular OS — but they're rather oblique.

   Solutions: Establish more and stronger incentives. For example, annual Free Software design awards could publicize and reward developers for good design. Software distributors could publish statistics on how many of their users use which programs, and how that number is changing over time. A bounty system could let people pay money in escrow for whoever implements a particular usability improvement. And distributed version control could foster quicker competition: distributors could choose not just which application to ship, but also which variant branch of an application, with usability as a factor in their choice.

2. **Few good designers.** Some musicians are also great composers, but most aren't. Some programmers are also great designers, but most aren't. Programming and human interface design are separate skills, and people good at both are rare. So it's important for software to have dedicated designers, but few Free Software projects do. Some usability specialists are employed by Free Software vendors such as Mozilla, Sun, Red Hat, and Canonical. But there aren't many, and skilled *volunteer* designers are even harder to find.

Solutions: Provide highly accessible training materials for programmers, and volunteer designers, to improve the overall level of design competence. Foster communities that let programmers collaborate with usability specialists. And encourage Free Software projects to have a lead programmer, a lead human interface designer, a help editor, and a QA engineer, these being separate people.

But why is there a shortage of volunteer designers in the first place? That brings us to the third problem.

3. **Design suggestions often aren't invited or welcomed.** Free Software has a long and healthy tradition of "show me the code". But when someone points out a usability issue, this tradition turns into "patches welcome", which is unhelpful since most designers aren't programmers. And it's not obvious how else usability specialists should help out.

   Solution: Establish a process for usability specialists to contribute to a project. For example, the lead designer could publish design specifications on the project's Web site, and invite feedback on a Weblog, wiki, or mailing list. The designer could respond courteously to design suggestions (even the misguided ones). And the project maintainer could set up an editable issue tracker, instead of an append-only bug tracker — making it easy to refine, approve or decline, and prioritize implementation of design suggestions in the same way as bug reports.

So why do programmers respond differently to usability suggestions than to more technical bug reports?

4. **Usability is hard to measure.** Some qualities of software are easily and precisely measured: whether it runs at all, how fast it starts, how fast it runs, and whether it is technically correct.

   But these are only partial substitutes for more important qualities that are harder to measure: whether the software is useful, how responsive it feels, whether it behaves as people expect, what proportion of people succeed in using it, how quickly they can use it, and how satisfied they are when they're finished.

   These human-related qualities can often be measured in user tests. But doing that takes hours or days that volunteers are unwilling to spend. User tests are usually low-resolution, picking up the big problems, but leaving designers without hard evidence to persuade programmers of the small problems. And even once a problem is identified, a solution needs to be designed, and that may need testing too.

   Without frequent user testing, volunteer projects rely on subjective feedback from the sort of people devoted enough to be subscribed to a project mailing list. But what these people say may not be representative of even their own actual behavior, let alone the behavior of users in general.

   Solutions: Promote small-scale user testing techniques that are practical for volunteers. Develop and promote screen capture, video recording, and other software that makes tests easier to run. Encourage developers to trust user test results more than user opinions. And write design guidelines that give advice on the common small problems that user tests won't catch.

The lack of dedicated designers, in turn, contributes to three cultural problems in Free Software projects.

5. **Coding before design.** Software tends to be much more usable if it is, at least roughly, designed before the code is written. The desired human interface for a program or feature may affect the data model, the choice of algorithms, the order in which operations are performed, the need for threading, the format for storing data on disk, and even the feature set of the program as a whole. But doing all that wireframing and prototyping seems boring, so a programmer often just starts coding — they'll worry about the interface later.

   But the more code has been written, the harder it is to fix a design problem — so programmers are more likely not to bother, or to convince themselves it isn't *really* a problem. And if they finally fix the interface after version 1.0, existing users will have to relearn it, frustrating them and encouraging them to consider competing programs.

   Solution: Pair up designers with those programmers wanting to develop a new project or a new feature. Establish a culture in Free Software of design first, code second.

6. **Too many cooks.** In the absence of dedicated designers, many contributors to a project try to contribute to human interface design, regardless of how much they know about the subject. And multiple designers leads to inconsistency, both in vision and in detail. The quality of an interface design is inversely proportional to the number of designers.

   Solution: Projects could have a lead human interface designer, who fields everyone else's suggestions, and works with the programmers in deciding what is implementable. And more detailed design specifications and guidelines could help prevent programmer-specific foibles.

7. **Chasing tail-lights.** In the absence of a definite design of their own, many developers assume that whatever Microsoft or Apple have done is good design. Sometimes it is, but sometimes it isn't. In imitating their designs, Free Software developers repeat their mistakes, and ensure that they can never have a *better* design than the proprietary alternatives.

   Solution: Encourage innovative design through awards and other publicity. Update design guidelines, where appropriate, to reflect the results of successful design experiments.

Other reasons for poor usability exist regardless of the presence of dedicated designers. These problems are more difficult to solve.

8. **Scratching their own itch.** Volunteer developers work on projects and features they are interested in, which usually means software that they are going to use themselves. Being software developers, they're also power users. So software that's supposed to be for general use ends up overly geeky and complicated. And features needed more by new or non-technical users — such as parental controls, a setup assistant, or the ability to import settings from competing software — may be neglected or not implemented at all.

   Solutions: Establish a culture of simplicity, by praising restrained design and ridiculing complex design. And encourage volunteer programmers to watch their friends and families using the software, inspiring them to fix problems that other people have.

9. **Leaving little things broken.** Many of the small details that improve a program's interface are not exciting or satisfying to work on. Details like setting a window's most appropriate size and position the first time it opens, focusing the appropriate control by default when a window opens, fine-tuning error messages and other text to be more helpful, or making a progress bar more accurately reflect overall progress. Because these things aren't exciting or satisfying, often years go by before they get fixed. This gives users a general impression of poor design,

and that may in turn discourage usability specialists from contributing.

Solution: When scheduling bug fixes, take into account how long they will take, possibly scheduling minor interface fixes earlier if they can be done quickly. Involve interface designers in this scheduling, to guard against usability flaws being downplayed because “it’s just a UI issue”.

10. **Placating people with options.** In any software project with multiple contributors, sometimes they will disagree on a design issue. Where the contributors are employees, usually they’ll continue work even if they disagree with the design. But with volunteers, it’s much more likely that the project maintainer will agree to placate a contributor by adding a configuration setting for the behavior in question. The number, obscurity, and triviality of such preferences ends up confusing ordinary users, while everyone is penalized by the resulting bloat and reduced thoroughness of testing.

    Solution: Strong project maintainers and a culture of simplicity. Distributed version control may help relieve the pressure, too, by making it easier for someone to maintain their own variant of the software with the behavior they want.

11. **Fifteen pixels of fame.** When a volunteer adds a new feature to a popular application, it is understandable for them to want recognition for that change — to be able to point to something in the interface and say “I did that”. Sometimes this results in new options or menu items for things that should really have no extra interface. Conversely, removing confusing or unhelpful features may draw the ire of the programmers who first developed them.

    Solutions: Provide alternative publicity, such as a Weblog, for crediting contributors. Establish design review of code changes that affect the human interface. Regularly review the entire interface, asking “Do we really need this bit”.

12. **Design is high-bandwidth, the Net is low-bandwidth.** Volunteer software projects are usually highly distributed, with contributors in different cities or even different continents. So project communications are mostly plain text, in e-mail, instant messaging, IRC, or a bug tracking system. But interaction design is multi-dimensional, involving the layout and behavior of elements over time, and the organization of those elements in an overall interface.

    When developers are in the same room, they can discuss interaction design using whiteboards, paper prototypes, spoken words, and gestures. But on the Internet, these often aren’t available, making discussions much slower and prone to misunderstandings.

INTERNET ARCHIVE
WayBack Machine
http://mpt.net.nz/archive/2008/08/01/free-software-usability
Go
JUL AUG SI
70 captures
5 Aug 08 - 19 Dec 13
2007 5 2008 20

Develop and promote VoIP, video chat, virtual whiteboard, sketching, and animation software that [illegible] net. Whenever possible, hold physical meetings for developers to collaborate in person.

Finally, a few problems are specific to Free Software development.

13. **Release early, release often, get stuck.** The common practice of “release early, release often” can cause poor design to accumulate. When a pre-release version behaves a particular way, and testers get used to it behaving that way, they will naturally complain when a later pre-release version behaves differently — even if the new behavior is better overall. This can discourage programmers from improving the interface, and can contribute to the increase in weird configuration settings.

Solution: Publish design specifications as early as possible in the development process, so testers know what to expect eventually.

14. **Mediocrity through modularity.** Free software hackers prize code reuse. Often they talk of writing code to perform a function, so that other programmers can write a "front end" (or multiple alternative "front ends") to let humans actually use it. They consider it important to be able to swap out any layer of the system in favor of alternative implementations.

    This is good for the long-term health of a system, because it avoids relying on any single component. But it also leads to a lack of integration which lessens usability, especially if the interfaces between the layers weren't designed with usability in mind.

    For example, most terminal commands do not provide information on how complete a process is, or estimate how much time is remaining. This is traditional behavior in a terminal, but in graphical software, progress feedback is crucial. If a graphical utility is developed merely as a "front end" to the terminal command, it can't easily provide that feedback.

    Solution: Design an example graphical interface first, so that interface requirements for the lower levels are known before they are written.

15. **Gated development communities.** When you do anything on a computer system you are relying on software from several different development teams. For example, if you print this Web page to show someone else, that task will involve not just the Web browser, but also a layout engine, a window manager, an interface toolkit, a variety of other libraries, a graphical subsystem, a printing subsystem, a printer driver, a filesystem, and a kernel, almost all of these implemented by separate teams.

    Oten these teams don't communicate with each other frequently. And unlike their proprietary competitors, they nearly all have different release cycles. This makes usability improvements difficult and slow to implement, if those improvements involve coordinating changes across multiple parts of the system.

    Solutions: Free Software system vendors can coordinate cross-component features like this, if they have employees working on all relevant levels of the software stack. And volunteer contributors to different software layers can meet at conferences arranged for that purpose.

That's a long list of problems, but I think they're all solvable. In the coming months I'll discuss examples of each of the solutions, and what I'm doing personally to help make Free Software a success.

Further reading

- "The usability of open source software" by David M. Nichols and Michael B. Twidale
- "Ronco spray-on usability" by John Gruber

This entry was posted on Friday, August 1st, 2008 at 11:50 pm and is filed under Computing & Internet, Usability.

« Ubuntu and "desktop environments"