What Makes Software High-Quality?

Shlomi Fish <shlomif@shlomifish.org>

Revision History

Revision 1675	02 April 2007	shlomif
Forked the template from a previous work and working on it.		
Revision 1871	04 May 2008	shlomif
Finalised the first draft. About to receive feedback.		
Revision 1872	04 May 2008	shlomif
Fixed many spelling/grammar/etc. problems, trailing whitespace, etc.		
Revision 1882	06 May 2008	shlomif
Fixed many spelling, grammar and syntax problems (thanks in part to Omer Zak). Added some bolds, and id=""s, added the about section, reorganised the text a bit, and added the note about responsiveness and startup time.		
Revision 1884	06 May 2008	shlomif
Changed "a software" to something more idiomatic, and fixed a bad phrasing towards the beginning.		

Abstract

This document will discuss what makes an open source program (and by induction other programs) high-quality. It will cover the parameters that make software applications high-quality and ways to achieve them.

Table of Contents

Introduction

What is a high-quality software application? I'm not talking about the application that's the most hyped, or the fastest, most featureful, or best. You can often hear endless rants and Fear, Uncertainty and Doubt (FUD) attacks about it. Often, it has competing programs that are superior in many respects. But at the end of the day, it is one of the most popular programs out there and is what many people like to use, what they find as the right tool for the job, and often what they are told (or even forced) to use.

How did I think about it? I am active in the fc-solve-discuss mailing list, which is dedicated to computerised techniques for solving the Freecell solitaire card game, and for discussing other automated solving and research of Solitaire games. This mailing list was started after I wrote Freecell Solver (with a lowercase "c" and an uppercase "S"), an open-source solver, which has proven to be quite popular. However, this mailing list was not restricted to discussing it exclusively, and was joined by many other Freecell solving experts, researchers and enthusiasts.

Later on, I lost interest in Freecell Solver, and have only been fixing bugs in the latest stable version (2.8.x). Since it is open source, and with an aim to be a Bazaar-like project I welcome other people to take over and am willing to help them, but it is still very functional as it is.

Recently, this message was written by Gary Campbell, who wrote:

> I think the solver discussion in the Wikipedia should mention that the FCPro solvers give quite lengthy and virtually unusable solutions. No human wants to follow the several hundred steps that usually result in order to get at what's really required to solve a given layout. If someone wants a solution that can be understood, they want one that is under 100 steps. This is a major contribution and it should be stated. There are a lot of **"toy"** solvers and only a very few of **"industrial strength."** One of the latter is the solver by Danny Jones. I don't see that on your list. The results he has gotten from his solver pretty much put your solver to shame. The more you hype Freecell Solver, the more criticism you open yourself up to.

Unfortunately, it's hard for me to determine what "industrial strength", "enterprise grade" and other such buzzwords are. But I'll try to define high quality here, and try to show when a program is high quality, and when it is not exactly the case.

I initially wanted to give some examples for software that I considered to be of exceptional high-quality, but I decided against it. That's because it is a matter of taste if this is the case for them, and since it may

provoke too much criticism against this essay as a whole. Thus, I'll just give some examples, possibly accompanied by screenshots, of places where one program does something better than a different program, while not even implying that the latter is in fact lower-quality in all possible respects.

Parameters of Quality

This section will cover the parameters that make software high quality. However, it will not cover the means to make it so. These include such non-essential things as having good, modular code [having_good_code] good marketing, or having good automated tests. These things are definitely important, but will be covered only later, and a lot of popular, high-quality software lacks some of them, while its competition may do better in this respect.

The Program is Available for Downloading or Buying

That may seem like a silly thing to say, but you'll be surprised how many times people get it wrong. How many times have you seen web-sites of software that claim that the new version of the software (or even the first) is currently under work, will change the world, but is not available yet? How many times have you heard of web-sites that are not live yet, and refuse to tell people exactly what they are about?

Alternatively, in the case of the "Stanford checker", which is a sophisticated tool for static code analysis, it is not available for download, but instead is a service provided by its parent company.

A software should be available in the wild somehow (for downloading or at least buying) so people can use it, play with it, become impressed or unimpressed, report bugs, and ask for new features. Otherwise it's just in-house software or at most a service, that is not adequate for most needs.

In the "Cathedral and the Bazaar", Eric Raymond recommends to "release Early and release Often". Make frequent, incremental releases, so your project won't stagnate. If you take your project and work on it yourself for too long, people will give up.

If you have a new idea for a program, make sure you implement some basic but adequate functionality, and then release it so people can play with it, and learn about it. Most successful projects that were open-source from the start have started this way: the Linux kernel, gcc, vim, perl, CPython. If you look at the earliest versions of them all, you'll find they were very limited if not downright hideous, but now they are often among the best of breed.

The Version Number is Clearly Indicated

Which version of your software are you using? How can you tell? It's not unusual to come to a page where the link to the archive does not contain a version number, nor is it clearly indicated anywhere. What happens if this number was bumped to indicate a bug fix. How do you indicate it then?

A good software always indicates its version number in the archive file name, the opening directory file name, has a `--version` command-line flag, and the version mentioned in the about dialogue if there is one.

[having_good_code] What? Shouldn't high quality software have a good codebase. Surprisingly no. What would you prefer: having a very modular codebase that does something pretty useless like outputting the string "Hello World" **or** having a large codebase of relatively low quality with a large amount of useful functionality and relatively few bugs?

I would certainly prefer the other alternative. That's because I know I can always refactor that codebase, either by large scale refactoring or by continuous refactoring or even "Just-in-Time" refactoring, only to add a new feature.

The Source is Available, Preferably under a Usable Licence

Public availability of the source is a great advantage for a program to have. Many people will usually not take a look at it otherwise, for many reasons, some ideological but some also practical. (Assuming there was ever a true ideology that was not also practical.) Without the source, and without it being under a proper licence, the software will not become part of most distributions of Linux, the BSD operating systems, etc.

If you just say that your software "is copyright by John Doe. All Rights Reserved", then it may arguably induce an inability to study its internals (including to fix bugs or add features) without being restricted by a non-compete clause, or even that its use or re-distribution is restricted. Some software ship with extremely long, complicated (and often not entirely enforceable) End-User-Licence-Agreements (EULAs) that no-one reads or cares to understand.

As a result, many people will find a program with a licence that is not 100% Free and Open Source Software - unacceptable. To be truly useful the software needs to also be GPL compatible, and naturally usable public-domain licences such as the modified BSD licence, the MIT X11 licence, or even pure Public Domain source code[public-domain], are even better for their ability to be sub-licensed and re-used. This is while licenses that allow incorporation but not sub-licensing, like the Lesser General Public License (LGPL) are somewhere in between.

While some software on Linux has become popular despite being under non-optimal licences, many Linux distributions pride themselves that the core system consists of "100% free software". Most software that became non-open-source, was eventually forked or got into disuse, or else suffered from a lot of bad publicity.

As a result, a high-quality software has a licence that is the most usable in the context of its operation. These licences are doubly-important for freely-distributed UNIX software.

It "Just Works"

This is a meta-parameter for quality. When people say that something "just works", they mean that you don't have to be concerned about getting it up and running, not spend too much time to learn it, not worry about it destroying data, or have to wonder how to troubleshoot problems with it.

A program that just works is the holy grail of high-quality software. In practice this means several things:

1. A "just works" software also doesn't have **any show-stopping bugs**. While it may still have some bugs, it should mostly function correctly.

2. It has most of the **features people want** and does not lack essential ones. For example, GNU Arch, an old and now mostly unused version control program, did not work on Windows 32-bit, while Subversion, a different and popular alternative, has a native port. Moreover, Mercurial, a different alternative, cannot keep empty directories in the repository. This may make both Mercurial and GNU Arch a no-starter for many uses.

[public-domain] Using a pure-public-domain Licence for your software is problematic because not all countries have a concept of "public-domain", similar to that of the United States, because many people misinterpret it, and because it is not clear whether software can be licensed under the public-domain to begin with. (And other such issues).

While quite a lot of important programs has been released under the public domain, and they are doing quite fine, they may have some problematic legal implications.

For these reasons, I now prefer the MIT X11 Licence for software that I originated instead of the "public domain".

Tendra is the most prominent alternative C and C++ compiler to GCC, but it's hardly as advanced as GCC is, does not have all of GCC's features and extensions, and is not usable as a replacement for GCC for most needs. As such, it is hardly ever used.

3. A "just works" software also has **good usability**. What it means is that it behaves like people expect it to. The Emacs-based editors, which are an alternative to Vim do not invoke the menus upon pressing "Alt+F", "Alt+E", etc. which is the Windows convention to them.

   Furthermore, when putting a single-line prompt, the prompt cannot be dismissed with either Ctrl+C or ESC, while in Vim, both keys dismiss the prompt. The key combination to dismiss it is not written anywhere on the screen and I won't tell you what it is. According to User Interface Design for Programmers, “A user-interface is well-designed when the program behaves exactly how the user thought it would”.

   While some people may be led to believe this is not applicable to terminal applications, TTY applications, command line applications, or even Application Programming Interfaces (APIs) - it still holds there. One thing that made me like gvim (the Graphical front-end to vim) was that it could be configured to behave much like a Windows editor. I gradually learnt more and more vim paradigms, but found the intuitive usability a great advantage. But I could never quite get used to Emacs.

The Program has a Homepage

Most software of high-quality has a homepage which introduces the software, has download information, links, etc. And no - a /project/myprogram/ page on Source Forge or whatever is much more sub-optimal than that, and leaves a bad impression.

The Software is Easy to Compile, Deploy and Install

A high-quality software is easy to compile, deploy and install. It builds out of the box with minimal hassles. There are several common standard building procedures for such software:

1. The common standard building procedure using the GNU Autotools is `./configure --prefix=$PREFIX ; make ; make install`.

2. There are now some more modern alternatives to the GNU Autotools, which may also prove useful.

3. CPAN Perl distributions have a similar `perl Makefile.PL` procedure or more recently also one using `perl Build.PL` which tends to be less quirky (see Module-Build).

   Generally, one usually installs them using the CPAN.pm or CPANPLUS.pm interfaces to CPAN, or preferably using a wrapper that converts every CPAN distribution to a native (or otherwise easy to remove) native system package.

4. Python packages have the standard `setup.py` procedure which can also generate Linux RPMs and other native packages.

5. There are similar building procedures for most other technologies out there.

However, it's not uncommon to find software that fails to build even on GNU/Linux on an x86 computer, which is the most common platform for development. Or the case of the qmail email server, which has a long and quirky build process. It reportedly fails to compile on modern Linuxes, and someone I know who tried to build it said that it did not work after following all the steps.

One thing that detracts from a piece of software being high-quality is a large amount of dependencies.

If we take Plagger, a web-feed mix-and-match framework in Perl (not unlike Yahoo Pipes, but predates it), then its Plagger distribution on CPAN contains all of its plug-ins inside, and as a result requires "half of CPAN" including such obscure modules, as those for handling Chinese and Japanese dates.

Popular programs like GCC, perl 5, Vim, Subversion and Emacs have very few dependencies and they are normally included in the package, if necessary to build the system. They are all written in very portable ANSI C and POSIX and have been successfully deployed on all modern UNIX-flavours, on Microsoft Windows and on many other more obscure systems.

While reducing the number of dependencies often means re-inventing wheels, it still increases the quality of your software. I'm not saying software cannot be high-quality if it has a large amount of dependencies, but it's still a good idea to keep it to a minimum.

The Software Has Packages for Most Common Distributions

A good program has packages for most common distributions, or such packages can be easily prepared.

Lack of such packages will require installing it from source, using generic binary packages, or other workarounds that are harder than a simple command to install the package from the package manager, and may prevent it from being maintained into the future.

A good example for how this can become wrong is the qmail SMTP server, before it became public-domain. The qmail copyright terms prevented distributing modified sources, or binary packages. As a result, the distributions that supported it packaged it as a source package, with an irregular build-process. Since the qmail package had its own unconventional idea of directory structure, some of the distributions had to extensively patch it. This in turn prevented more mainstream patches from being applied correctly to correct the many limitations that qmail had, or accumulated over the years due to its lack of maintenance.

The Software Has Good, Extensive and Usable Documentation

If your GUI program is simple and well-designed, then you normally don't need good documentation. However, a command line program, a library, etc. does need one, or else the user won't know what to do.

There are many types of documentation: the --help flag, the man page, the README/USAGE/INSTALL files, full-fledged in-depth guides, documents about the philosophy, wikis, etc. If the program is well-designed, then the user should be able to get up and running quickly. An exception to this is various highly specialised programs, such as 3-D graphics programs, or CADs, that require some learning.

If we take Subversion as an example, then it has a full Book online, several tutorials, an `svn help` command which provides help to all the other commands, and a lot of help can be found using a Google search. GNU Arch, on the other hand, only had one wordy tutorial, that I didn't want to read. Most of the other tutorials people wrote, became misleading or non-functional as the program broke backwards compatibility.

Vim has an excellent internal documentation system. It's the first thing you are directed see when invoking it. It has a comprehensive tutorial, a full manual, and the ability to search for many keywords, with a lot of redundancy. As a result, one can easily become better and better with vim or gvim, albeit many people can happily use it with only the bare essentials.

Emacs' help on the other hand is confusing, dis-organised, lacking in explanation and idiosyncratic. It doesn't get invoked when pressing "F1", is not directed to when the program starts, and most people cannot make heads nor tails of it. There is a short Emacs tutorial, but it isn't as extensive as Vim's. Nor does

it explain how to configure Emacs to behave in a better way than its default way in which it behaves completely different to what people who are used to Windows-like conventions or vim-like conventions expect.

Portability

It is tempting to believe that by writing a program for one platform, you can gain most of your market-share. However, people are using many platforms on many different CPU architectures: Windows 32-bit/64-bit on Intel machines, Itanium, or x86-64; Linux on a multitude of platform; BSD systems (NetBSD, FreeBSD, OpenBSD and others) on many architectures as well; Mac OS X; Sun Solaris (now also OpenSolaris), and more obscure (but still popular) Unix-clones like AIX, HP-UX, IRIX, SCO UNIX, Tru64 (formerly Digital Unix), etc. And to say nothing of more exotic, non-UNIX, non-Microsoft operating systems like Novell Netware, Digital Corp.s's VMS or OpenVMS, IBM's VM/CMS or OS/390 (MVS), BeOS, AmigaOS, Mac OS 9 or earlier, PalmOS, VxWorks, etc. etc.

As a general rule, the only thing that runs on top of all of these systems (in the modern "All the world is a VAX." world) is a C-based program or something that is C-hosted. [non-vax-like]Most good programs are portable to at least Windows and most UNIXes and potentially portable to other platforms.

For example, Subversion has made it a high priority to work properly on Windows. On the other hand many of its early alternatives, especially GNU Arch, could not work there due to their architectures. As a result, many mixed shops, Windows-only shops, or companies where some developers wanted to use Windows as their desktop OS, could not use Arch. So Arch has seen a very small penetration.

The bootstrapping ANSI C compiler of gcc for example, is written in very portable K&R C, so it can be compiled by any C compiler. Later on, this compiler can be used to compile most of the rest of the GCC compilers.

Compare that to many compilers for other languages that are written in the same language. For example, GHC - The Glasgow Haskell Compiler is written in itself, and requires a modern enough version of itself to compile itself. So you need to bootstrap several intermediate compilers to build it.

Security

A high-quality software is secure. It has a relatively small number of security issues, and bugs are fixed there as soon as possible.

Some people believe that security is the most important aspect of software, but it's only one factor that affects its quality. For example, once I was talking with a certain UNIX expert, and he argued that the Win32 CreateProcess() system call was superior to the UNIX combination of Fork() and Exec(), just because it made some bugs harder to code. However, some multitasking paradigms are not possible, without the fork() system call, which is not present in the Win32 API at all, and needs to be emulated (at a high run-time cost) or replaced with thread-based multitasking, which is not identical. Finally, it is still possible to get fork()+exec() right, and there's a spawn() abstraction on many modern UNIXes.

While I don't mean you shouldn't pay attention to security, or keep good security practices in mind when coding, I'm saying that it shouldn't slow down the process by much, or prevent too many exciting features from being added, or cause the development to stagnate.

[non-vax-like] I'm fully aware that before C-based UNIX and UNIX-like systems became dominant there were some more exotic architectures that could not run C comfortably. Prime examples for them are the PDP-10 and the Lisp machines.

However, such more "unconventional" architectures are now dead, and no CPU architecture developer in their right mind would want to create a CPU that won't be able to run C and C-based UNIX-based or UNIX-like operating systems such as Linux. (Unless it's probably a relatively niche micro-processor for embedded systems).

Lisp, and similar higher-level languages, run on modern UNIX-based OSes very well, so there's not a big problem there.

Backwards Compatibility

A high-quality program maintains as much backward compatibility with its older versions as possible. Some backward compatibility, like relying on bugs or other misbehaviours ("bug-to-bug compatibility"), is probably too extreme to consider. But users would like to upgrade the software and expect all of their programs to just continue to work.

A bad example for software that does not maintain backwards compatibility is PHP, where every primary digit breaks the compatibility with the older one: PHP 4 was not compatible with PHP 3 and PHP 5 was not compatible with PHP 4. Furthermore, sometimes existing user-land code was broken in minor-digit releases. As such maintaining PHP code into the future is a very costly process, especially if you want it to work with a range of versions.

On the other hand, the perl5 developers have been maintaining backwards compatibility between 5.000, 5.001, 5.002 up to 5.6.x, 5.8.x and now 5.10.x. As such one can normally expect older scripts to just work. perl5 can also run a lot of perl4 code and below, and perl4 code can be ported to modern versions of perl5 with some ease. While sometimes scripts broke (due to lack of "bug-to-bug compatibility"), or became slower, upgrading to a new version of Perl is normally straightforward.

Good Ways to Provide Support

A high-quality software has good ways for its users to receive support. Some examples for ways to do that are:

1. A Mailing List.
2. IRC (Internet Relay Chats) Channels.
3. An email address for questions.
4. Web Forums.
5. Wikis.

Without good ways to receive support, users will be unnecessarily frustrated when they encounter a problem, which cannot be answered by the documentation. Refer to Joel Spolsky's "Seven Steps to Remarkable Customer Service" for more information on how to give good support.

Speed and High Performance

The reason I mentioned this quality parameter so late is because it was what Mr. Campbell stressed in his argument about "Industrial Strength" Freecell solvers. So I wanted to show that there are other important parameters beside it. However, raw performance is important, too.

If a program is too slow, or generates sub-optimal results, most people will be reluctant to use it and find using it daunting. They will either give up waiting for it to finish, or get distracted. If the output results of the program are too sub-optimal (assuming there's a scale to their optimality), then they will probably look for different alternatives.

As a result, it is important that your software will run quickly, and will yield good results. There are many ways to make code write faster, and covering them here is out of the scope of this article.

A good example for how such optimisations can make such a huge difference are the memory optimisations done to Firefox between Firefox 2 and Firefox 3, which greatly improved its performance, memory consumption, and reduced the number of memory leaks. It should be noted that often, reducing memory consumption can yield better performance because of a smaller number of cache misses, process memory swapping, and other such factors.

There are several related aspects of performance, that also affect the general quality and usability of a program. One of them is responsiveness, which often manifest as people complain that the program is "sluggish". Java programs are especially notorious for being such, for some reason, while programs written in Perl, Python, are more responsive and feel snappy, despite the fact that their backend are generally slower than the Java virtual machine.

A tangential aspect is that of startup time. Many programs require or have required a long time to start, which also makes using them frustrating, even if they are later responsive and quick.

Aesthetics

A good software (or web site) or other resource is aesthetically pleasing. Aesthetics in this context, does not necessarily mean very "artsy" or having a breath-taking style. But we may have run into software (usually one for internal use or one of those very costly, bad-quality, niche, software) that seemed very ugly and badly designed, with a horrible user-interface, etc.

Different types of applications, and those running on different platforms, have different conventions for what is considered aesthetic. In The Art of UNIX Programming, Eric Raymond makes the case for the "Silence is Golden" principle of designing UNIX command-line interfaces. Basically, a command line program should output as little as possible. Now observe the behaviour of aptitude (a unified interface for package management) on Ubuntu Gutsy Gibbon, when trying to install a non-existing package name:

```
root@shlomif-desktop:/home/shlomif# aptitude install this-does-not-exist
Reading package lists... Done
Building dependency tree
Reading state information... Done
Reading extended state information
Initializing package states... Done
Building tag database... Done
Couldn't find any package whose name or description matched "this-does-not-exist"
The following packages have been kept back:
  firefox firefox-gnome-support
0 packages upgraded, 0 newly installed, 0 to remove and 2 not upgraded.
Need to get 0B of archives. After unpacking 0B will be used.
Reading package lists... Done
Building dependency tree
Reading state information... Done
Reading extended state information
Initializing package states... Done
Building tag database... Done
```

15 lines of output, and only one of them in the middle is the informative one. Why is all this information a concern of mine, especially given the fact that they are all given in the same monotonous default colour.

On the other hand, here's what urpmi (a similar package management interface for Mandriva) says on Mandriva Cooker:

```
[root@telaviv1 ~]# urpmi this-does-not-exist
No package named this-does-not-exist
```

```
[root@telaviv1 ~]#
```

Exactly one line and it's informative. While aptitude certainly has its merits, its verbosity still makes it much more painful to use than urpmi, when I have to work on Ubuntu.

Conclusion

There are probably several parameters for software quality that I'm missing. However, the point is that one should evaluate the general quality of the software based on many parameters and not exclusively "security" or "speed" or whatever.

For example, many proponents of BSD operating systems claim that the various BSDs are superior to Linux because they are more secure, or because they are (supposedly) faster or are easier to manage, because their licence is less problematic than the GPL, etc. However, they forget that Linux has some advantages like being more popular (and so one can get support more easily), or like the fact that its kernel supports much more hardware, or that it has better vendor acceptance, and because more software is guaranteed to run with less problems on Linux than on the BSDs. [linux-bsd-soft]

I'm not saying the BSDs are completely inferior to Linux, just that Linux still has some cultural and technical advantages. Quality in software is not a linear metric, because it is affected by many parameters. If you're a software developer, you should aim to get as many of the parameters I mentioned right.

Ways to Achieve Quality

Now that we've covered the elements that make programs high-quality here's a non-exhaustive list of ways to actually achieve it. None of them are absolutely necessary for achieving the quality, but they certainly help a lot.

Many people often confuse them with quality. "This software has God-damn awful code, so it's a pile-of-crap." Well, what do the users care how bad the code is, as long as it is functional, has all the necessary features, and is (mostly) bug-free? It's important not to confuse quality with the ways to achieve it.

The aim of this section is to briefly cover as many measures for achieving good quality as possible.

Having Modular, Well-Written Code

The more modular a project's code is, the easier it is to change it, understand it and extend it, and the faster development will take. Refactoring is the name given to the process used to transform code from sub-optimal and "ugly", but still mostly functional and bug-free code, to a code that is equally functional but more modular and clean. See the "Joel on Software" excellent article "Rub a dub dub" for some of the motivation and practices of good refactoring instead of throwing away the code and restarting from scratch.

Automated Tests

Automated tests aim to test the behaviour of code automatically, instead of manual testing. The classical example for them is that if we wish to write a function called `add(i, j)` that aims to add two integers then we should check that `add(2, 3) == 5`, that `add(2, 0) == 2`, that `add(0, 2) == 2`, that `add(5, -2) == 3`, that `add(10, -24) == -14`, etc.

[linux-bsd-soft] Naturally, this is a problem with the fact that most developers are developing on Linux (mostly x86), don't test it on other Unix flavours, and are too careless or unaware to write their programs portably enough.

However, it's still a quality parameter, because it still affects the way you're using the operating system.

Then we can run all the tests and if any of them failed, we can fix them. Then after we write or modify the code, we can test using them again to see if there are any regressions.

Writing automated tests before we write the actual code, or before we fix a bug, and accumulating such tests (the so-called "Test-driven development" paradigm), is a good practice which helps maintain a high-quality software, and facilitates refactoring and makes it safer.

Beta Testers

If you have beta testers for the code, or publish development versions frequently, you can get a lot of feedback for various different platforms and configurations your code is running on. These beta-testers can run the automated tests and also use the beta-code for their own testing or even production.

Frequency of Releases

The more frequent your releases are, the more people can test your code, and the more they can upgrade to the latest version, and the quicker bugs that disturb your users are fixed, etc.

Naturally, there are advantages for slower release cycles, or for predictable release cycles like GNOME 2.x has. I won't voice a definite opinion for which is the best methodology, but such a decision should be taken into consideration.

Good Software Management

There are several sources online and offline explaining good software management for "shrinkwrap" software (open-source, commercial or other distributed) and for other types of software development (embedded, in-house, etc.), from which good advice can be taken for how to best run a software project. While the are sometimes contradictory, and often false, they still make a good read and are thought-provoking.

Here are some links:

1. Eric Raymond's "The Cathedral and the Bazaar" series
2. The "Joel on Software" site
3. Paul Graham's Essays (on various topics)
4. Extreme Programming

Good Social Engineering Skills

It certainly helps for the project's communities to have good social engineering skills. From tactfulness, to humour, to controlling one's temper, to saying "Thanks" and congratulating people for their contribution, to timely applying patches and fixing bugs - all of these makes contributing to a project and using the program more fun and less frustrating.

Often, social engineering should be made part of the design of the software, or the web-sites dedicated to it. For example, it took me several iterations of having to fill the same project form in the GNU Savannah software-hub, only for it to be rejected, and me having to follow the process again. Despite the fact the admins were polite, it still was annoying.

Eventually, they implemented a way to save previous project submissions and to re-send them, so future users won't become as frustrated as I did.

Again, some projects have succeeded despite the fact they had, or even still have, bad social engineering. But adopting a good software engineering policy can certainly help a lot.

No Bad Politics

Bad politics in a software project is a lot like subjectivity, it can never be fully eliminated, but we should still strive to reduce it to the minimum.. If bad political processes become common in a project, then important features are dropped, bugs are left unfixed, patches are stalled, external projects gets stalled or are killed, people become frustrated or angry and possibly leave the project - and the project may risk forking.

So it's a good idea to keep bad politics at bay and to a minimum. How to do that is out of the scope of the document, but it's usually up to the leaders to keep it so and maintain a good policy that will make as few people as possible frustrated and will not flabbergast external contributors. And naturally, for open-source and similar projects, a fork is often an option in this case, or in other similar cases.

Good Communication Skills

A project leader and other participants should have good communication skills: very good English; pleasantness and tact; good phrasing, proper grammar, syntax, phrasing and capitalisation; clear writing; patience and tolerance; etc. If he doesn't, then the project may encounter problems as people will find the project's developers hard to understand or tolerate, and, thus, hard to work with.

Hype

Despite common belief I believe that the less hype and general noise there is regarding a software project, the better off it is. For example, as Paul Graham notes regarding Java:

> [Java] has been so energetically hyped. Real standards don't have to be promoted. No one had to promote C, or Unix, or HTML. A real standard tends to be already established by the time most people hear about it. On the hacker radar screen, Perl is as big as Java, or bigger, just on the strength of its own merits.

In fact, I can argue that if your project receives a lot of negative hype, then it is an indication that it is successful. Perl, for example, has received (and still receives) a lot of criticism, and Perl aficionados often get tired of constantly hearing or seeing the same repetitive and tired arguments by its opponents. However, the perl 5 interpreter is in good shape, it has many automated tests (much more than most other competing dynamic languages), an active community, many modules on CPAN (the Comprehensive Perl Archive Network) with a lot of third-party , open-source functionality, and relatively few critical bugs. It is still heavily actively used and has many fans.

Similar criticism has been voiced against the Subversion version control system, Linux, etc. One thing one can notice about such highly-criticised projects is that they tend not to be bothered by it too much. Rather, what they say is that "If you want to use a competing project, I won't stop you. It probably is good. It may be better in some respects. I like my own project and that's what I'm used to using and use."

This is by all means the right policy of "hyping" to adopt, if you want your project to be successful based on its own merits. Some projects compete for the same niche, without voicing too much hype against each other or for them, and this is a better indication that they are all healthy.

A Good Name

Finally, a project should have a good name. One example for a project with an awful name is CVSNT. There are two problems with the name:

1. It is based on CVS, which most people have ran into its limitations, is considered passé and unloved, and people would rather avoid.

2. The "NT" part implies it only runs on Windows-NT, which is both misleading and undesirable.

On the other hand, the competing project "Subversion" has a much better name, since it has nothing to do with CVS, or Windows NT, and since it is an English word and sounds cool.

Some projects are successful despite being badly named, while some have a very cool name, but languish. Still, a good name helps a lot.

Also consider what Linus Torvalds said about Linux and 386BSD (half jokingly):

> No. That's it. The cool name, that is. We worked very hard on creating a name that would appeal to the majority of people, and it certainly paid off: thousands of people are using linux just to be able to say "OS/2? Hah. I've got Linux. What a cool name". 386BSD made the mistake of putting a lot of numbers and weird abbreviations into the name, and is scaring away a lot of people just because it sounds too technical.

Conclusion

Entire books (and web-sites) were filled with the various measures to achieve software quality, and what I described here was just a sample of it. The point is that these are not aspects of quality by themselves, but rather measures that help. None of them is a required or adequate condition for the success of a project, but the more are implemented the easier, faster, and more enjoying working on the project will be.

Analysis of the Quality of the Various Freecell Solvers

Introduction

As you recall, Gary Campbell's comment provided the motivation for my investigation into what makes a project high-quality and which quality-increasing-measures are not elements of quality by themselves. And since Freecell Solver has been my pet project, and as I'm still interested in techniques for solving Solitaire, I'd like to conclude this essay by analysing the various solvers according to the parameters I described. The natural caveat is that due to the fact I authored a solver of my own, then I may be a bit biased.

I was not so sure including this section is a good idea, but I feel the article is incomplete without it. Feel free to skip it, in case you are not that interested in it.

The Analysis Itself

As is not uncommon in many software niches, some Freecell solvers are **not commonly available** for downloading or even for buying. For example, Bill Raymond's solver (named "Cat in the Sack") which he has been talking about extensively, has not been released yet, under any licensing terms. Similarly, Danny A. Jones has kept his solver (mentioned in the introduction) for himself, and has not released it to the wild in either source or binary forms.

As such, the utility of such solvers is heavily reduced. For once, they cannot be used as integrated solvers of Solitaire apps, because no one is going to wait for an email to the author to be sent, and the author to reply with the solution. Also, many researchers won't trust solutions given by them without the ability to inspect the source code.

Most other solvers are **available for free download**, and many are accompanied with the source, most probably under an open-source licence.

Gary Campbell's Solver is written in 8086 Assembly, with some 32-bit extensions, and only runs on DOS and DOS-compatible platforms. The assembly is compiled using a self-hosting macro assembler written by

Campbell. The Assembly source code of the solver or the assembler are not available. As such modifying the code of the executable may prove to be problematic.

It is tempting to think that x86 and DOS are a very low common denominator, but that is not always the case. Imagine that you are writing a Freecell game for an embedded device running a non-x86 architecture such as ARM or PowerPC. In that case, in order to run Campbell's solver, you'll need to embed an x86-and-DOS emulator (such as DOSBox) inside the device, which will complicate things, consume a lot of memory, and slow things down. On the other, an ANSI C-based solver, such as Freecell Solver, or Tom Holroyd's Patsolve, can be easily made to compile and run there with few if any modifications. So they would be preferable.

Similarly, when writing **portable programs** for the various Unix flavours and other POSIX-capable or ANSI C-capable operating-systems, then writing code in x86 Assembly exclusively will make it a non-starter. [Portions of Assembly]

Some Freecell solvers also don't build out of the box. For example, typing `make` inside the patsolve distribution yields the following output:

```
shlomi:~/patsolve-3.0$ make
make clean
make[1]: Entering directory `/home/shlomi/patsolve-3.0'
rm -f patsolve *.o param.c param.h core win .depend
make[1]: Leaving directory `/home/shlomi/patsolve-3.0'
touch .depend
param.py param.dat
make: param.py: Command not found
make: *** [param.h] Error 127
```

This is relatively easy to fix, but still frustrating. Freecell Solver on the other hand, has been fully converted to the GNU Autotools, and can be built as a static and shared library (a.k.a DLL), as well as several proof-of-concept, but still usable, command line utilities that link against it.

Freecell Solver has very good **usability**: it mostly respects the Unix conventions and best practices, it can start solving from any arbitrary Solitaire board position given to it as input, and has kept command line interface usability in mind. Campbell's solver on the other hand, as of this writing, does not operate on a standard input/standard output manner, is poorly documented, and is counter-intuitive for someone who is used to Unix conventions (and most DOS conventions). I'm not even sure it can accept any arbitrary board as input, but I may be wrong.

Regarding **the homepage** of the solver: the Freecell Solver homepage has many pages, a navigation menu, a common look-and-feel and many links and information. On the other hand, Patsolve is nothing but a link in Tom Holroyd's software archive, which can easily be missed. Gary Campbell's homepage for his solver has a Baroque design and quite a lot of marketing-speak. And it's only one page with no anchors or a navigation menu, and very few links. Most other solvers don't fare better than that, and certainly worse than my own.

Licensing, a necessary, but important, evil. The Freecell Solver's ANSI C source is distributed under the Public Domain licence, which allows virtually any use, including linking, modifying and sublicensing

[Portions of Assembly] In some cases, the developers of portable software maintain versions of parts of the code in Assembly of selected architectures, while keeping more portable versions written in C or C++. This is done to optimise some parts for common CPU architectures.

under any different licence by a third party.[PD-LICENSE] On the other hand, Patsolve is distributed under the GPL licence which while usable is considerably more restrictive than Public Domain, or BSD-style licences. This is also the case for this Common Lisp solver. Many other solvers, including Campbell's are binary-only, "All Rights Reserved", which makes them a complete no-starter for most open-source applications out there. Furthermore, many non-open-source applications will prefer to use a Public Domain solver, rather than paying royalties or risking "copyrights" and source availability problems.

Freecell Solver (FCS) is extensively documented, and even its online help is helpful and usable by itself. The other solvers are much less documented, but arguably they also have much fewer options and features than Freecell Solver does.

Which brings us to the features - Freecell Solver has a long list of features, and I'm not aware of any solver with so many of them. Especially of note is that it can solve many other Solitaire variants, including Simple Simon, of which FCS is probably the only solver capable of solving.

Most of the solvers out there are fast, but some are more than others. Normally, for solving an individual game on the command-line, almost any solver will do, and the only cases where such speed matters more is for Solitaire research, and for analysing large sets of different games. (see the Freecell FAQ for more information).

Conclusion

All things considered, I still feel that Freecell Solver is probably the best quality Solitaire solver out there. While it still has a lot of room for improvement, the rest of its competition have much bigger issues, or have not been made available (yet or ever).

Several factors contributed for its success: the fact that I announced many releases on Freshmeat.net, that I received a lot of input from my users and co-developers, that I was determined to constantly improve it and work on it, and that I worked on creating and maintaining a good web-site and documentation.

While most of the contributions of code I received were limited, the input from the users of the software proved to be crucial for its prosperity. Like I noted earlier, I lost interest in working on it (at least temporarily), but still maintain it, and feel that it is good enough as it is.

I hope this document, and similar resources it referenced will help you in working on your software, and improving its quality for the benefit of your users and you.

Happy Hacking!

About This Document

Copyrights

It was written by Shlomi Fish who also holds the copyrights.

[PD-LICENSE] As noted earlier, a Public Domain non-licence has some problems, which may make the software problematic for many corporations and in some jurisdictions. However, I don't have problem in exempting the licence of the Freecell Solver code (at least not the Public Domain code that I fully originated), from the Public Domain, including under the MIT X11 Licence, assuming this is necessary.

I'm also considering relicensing Freecell Solver and other older software of mine under the MIT X11 licence, or possibly having a dual-Public Domain and MIT X11 licensing terms (which seems somewhat silly, but may be a good idea.)

About the Author

Shlomi Fish was born in Israel in 1977, and has lived there most of his life. He is a user, developer, advocate and activist of Open Source Software. His greatest single contribution so far in this area has been Freecell Solver, a Public Domain Library for solving games of Freecell and other types of Solitaire. However, he also initiated several other projects and made important contributions to projects he did not initiate some of them very large scale. Recently, most of his contribution were perl distributions he uploaded to CPAN under his username.

Fish's work on Freecell Solver and other open source projects, and the fact he is interested in many software management techniques has led him to write a large number of articles, and weblog entries which describe his unique views as a developer of mostly portable software.

Acknowledgements

Thanks to Mr. Gary Campbell, the author of "FCELL.COM" whose original message provided the inspiration for this article. Thanks to Omer Zak, who went over an early draft of this article and supplied many useful comments. Thanks to many people on the IRC (= Internet Relay Chat) who have read the article and comments, and chose not to be explicitly credited.

Finally, thanks to all the past and present people who helped me with Freecell Solver, and with the rest of my "Free and Open Source Software" (= FOSS) endeavours.