# Writing Quality Software

**About this white paper:** This whitepaper was written by David C. Young, an employee of General Dynamics Information Technology (GDIT). Dr. Young is part of a team of GDIT employees who maintain, and support high performance computing systems at the Alabama Supercomputer Center (ASC). This was written in 2020.

This paper is written for people who want to write good software, but don't have a master's degree in software architecture (or someone managing the project who does). Much of what is here would be covered in a software development practices class, often taught at the master's degree level.

Writing quality software is not only about the satisfaction of a job well done. It is also reflects on you and your professional reputation amongst your peers. In some cases writing quality software can be a factor in getting a job, losing a job, or even life or death. Furthermore, writing quality software should be considered an implicit requirement in every software development project. If the intended useful life of the software is many years, that is yet another reason to do a good job writing it.

## Introduction

Consider this situation, which is all too common. You have written a really neat piece of software. You put it out on github, then tell your colleagues about it. Soon you are bombarded with a series of complaints from people who tried to install, and use your software. Some of those complaints might be;

- It won't install on their version of Linux.
- They did the same thing you reported, but got a different answer.
- They used your example input, but got a different result.
- The prerequisites wont install on some operating system.
- It is incompatible with some of the prerequisites.
- It seems to have prerequisites that you didn't think it needed.
- It gets errors that seem like hardware errors.
- They have problems that you are guessing are user error, but you can't figure out what they are doing wrong.
- The program works, but only with certain inputs entered in a specific way. (This is called 'brittle software'.)
- They can't get it to work with some critical piece of their infrastructure, such as a job queue system.
- They can't import data to it, or export data from it.
- They can't seem to figure out how to use your software, or use it correctly.

A few weeks ago, you were so proud of this software. You saved hundreds of lines of code by calling other tools. You gave it some great functionality. You came up with a better way to do a common task. What is wrong? Is everyone else just incompetent? Probably not. Most likely you fell into a common programming trap of doing things that are 'smart but not wise'. In other words, you made software, but you didn't make quality software.

Software development is not just one task. It encompasses a whole range of tasks including architecting software, writing software, testing software, debugging, documenting, and coordinating the work of a team of software developers. Writing quality software is systemic task, meaning that it connects with all of these tasks. This document is a broad overview of the whole quality software development process.

## The source of software quality problems

Software quality is not just one thing. Here are some aspects of software quality, and their definitions in a software development context.

- Compatibility is the ability of software to work correctly with other software or hardware.
- Software robustness is often defined as handling incorrect inputs gracefully.
- Another definition of "robust software" is
    1. Easily installed on multiple computers
    2. Consistently works as advertised
    3. Can be integrated with other tools
- Software reliability refers to consistently getting correct answers, not corrupting your data, and keeping this behavior with environmental issues (i.e. network interruptions).
- Software stability means that the program doesn't crash, doesn't have memory leaks, and continues to function with a changing environment around it (i.e. installing a new operating system version).

**Dependencies:** There are many function libraries and modules that might be a great benefit to your software development project, but not all of them are quality software. If you work hard to write quality software, then your software calls a poor quality library, you have already failed.

There are some tell tail signs that the library your software is calling is quality software. Is it well documented? Is it developed based on a principle of back compatibility? Do many other programs depend on it, particularly ones you consider to be quality software? Do those programs document that they work with version X and newer of this library? Is the software design object oriented, or mimicking object oriented design in a non-object language? Another good sign is if their version numbering system is set up so versions within the same major release are compatible. If it is quality software, all or most of these will be true.

The sheer number of dependencies is also a concern. If your program has a large number of dependencies, how can you ensure that many versions of those dependencies are all compatible with your program, and with each other, and with many operating systems, and with multiple compilers? Software becomes more robust with better compatibility as the number of dependencies decreases.

**Compatibility:** One type of compatibility to strive for is input/output compatibility, such as file format compatibility. Sometimes multiple programs say they work with the same file format, but they can't always exchange those files. A good option for ensuring file compatibility is for all of the programs that use that file type to do their file access using the same library. You should document which version of that library you used in the developer documentation, and the installation directions.

Compatibility with dependent libraries, or a lack thereof, is one of the most common sources of software installation problems. As described above, choosing which libraries to make your software dependent on is part of that equation. However, there are some best practices for choosing which version of a library to use, as described below.

Compatibility with hardware can be mitigated in several ways. First, is using a separate hardware driver, such as OFED to access InfiniBand hardware. A second import concern is instruction set compatibility with the processor chip. The default compiler optimization flags (-03) compile software optimized for a Pentium 4 processor with 128 bit SSE instructions. This is done because that was the last chip where Intel and AMD chips were 100% instruction set compatible. Your software will run faster if you use complier flags for the actual processor in your computer, but now the executable may not run on the other brands and models of computer. One option is to distribute multiple complied versions of the program. Another option is to distribute source code, along with documentation and build utilities for selecting the appropriate compilation flags.

Operating system compatibility is another variation on library compatibility. Developing software that is easy to install and run on Windows, Linux, and Macintosh is not a happy accident. It takes planning, testing, and diligence on the part of the software development staff to make this dream come true. A number of suggestions are in the following section of this paper. If you admit defeat and have developed software that only works on one operating system version, distribute your software in a container, perhaps on Docker Hub or Singularity Hub. Docker and Singularity containers can be run on many different operating systems.

**Testing:** Novice programmers assume that software works if it compiles correctly. Experienced programmers assume software doesn't work correctly unless they have a test to prove it does. There are dozens of types of software tests. If you can only name two, you need to expand your software testing skill set.

**Configurability:** A lack of configurability is not equivalent to ease of use. Software ease of use is great, but there are right ways and wrong ways to create it. If your program always looks for a genome file in a particular directory, it may be tempting to hard code that path in the program. However, you can't be sure the same directory will be an option when someone else installs the software on their computer. A better option is to make a configurable setting for where to find the file.

**Documentation:** If you didn't write documentation, you didn't correctly write software. Even if your company has a documentation department, the people writing the software will have to write the first draft and make editorial corrections. Good documentation includes describing the program functions, giving examples or tutorials, and writing a programmers guide to help new software developers get started. Good documentation tells you when and why you would choose an option, not just what it does.

## Software quality best practices

The following are some more detailed examples of quality software development practices. Not every software development project will use all of these, but most projects should be using most of these.

**Plan for operating system version compatibility.** Look up studies on which versions of the operating system, compiler, and language are in common use. Then choose which you plan to support to ensure your software will be usable by at least 90% of your potential user base. It is best to do this before you even begin writing code, but it is better late than never.

**Plan for cross platform/compiler/library compatibility at the beginning of the project.** It's a lot easier to start out with a cross-platform process than to deal with it later. One option that is becoming increasingly viable is to plan to develop on one operating system (i.e. Linux) then to distribute to others (Macintosh, Windows, Unix) via a container.

**Rely on existing build tools and package managers for installation.** If you write your own compile and installation script instead of following a standard process like the GNU configure-make process, you are just going to create problems and generate animosity from every professional who tries to install your software. A little bit of time spent learning to use standardized tools like make or cmake pays off for years to come.

**Distribute and use static linked executables.** Yes, compilers default to making dynamic linked executables. That is a good idea for utilities in an operating system because it saves a bit on disk space, and decreases the memory needs of your program slightly. However, dynamic linked executables are far more prone to running into compatibility issues with the libraries they are linking. In the case of the executables shipped with the operating system, the developers do massive amounts of testing to ensure compatibility, and they only have to test on one operating system. For someone building an applications package, static linking will let it run on many operating systems (i.e. every 64 bit Linux distribution) with few compatibility issues. Also, static linked programs execute a bit faster than dynamic linked programs.

Unfortunately, there are some libraries (i.e. MPI) which cannot be static linked, so this isn't always an option.

**Develop with older versions of compilers, languages, and libraries.** If a library is designed to be back compatible, you can do the software development work with a somewhat older version (within reason) and your software should work with every version from the one used in development and newer. If you find that the program must have a particular version or newer to avoid a bug in the older versions or access needed functionality, document that fact in your developer documentation, in the installation documentation, and in the system requirements that users see before they download the software. Some libraries will even force you to use a compiler version that is not the most recent (i.e. the CUDA library and compiler).

**Develop on multiple platforms every day.** Do not write code on one operating system, then plan to port to other platforms later. Do not outsource the cross-platform part of the development. Virtual machines and cloud services can be convenient ways to test on multiple types of hardware.

**Make your development staff move from platform to platform.** That way the whole thing doesn't fall apart when you lose the one person who did the Windows work and didn't document their kludgy process.

**Enforce your coding practices.** Sometimes the best move you can make is to fire a programmer who refuses to follow the practices you have established for a good reason.

**Standards:** Standards are designed to make software more portable and compatible. Develop your software with an older version of the most non-enhanced implementation of the standard (i.e. develop MPI software using an older MPICH version). Likewise, use Unicode for text.

**Follow conventions.** Making software use MPI without a standard MPI launcher is asking for compatibility issues.

**Be wary of third-party application frameworks or runtime environments.** These claim to save you effort, but a large amount of effort goes into learning to use them, then they become a new source of errors. There are situations where one of these is so well written (and probably dominant in its field) that it can be a good choice. However, assume these are a bad decision until you have tested it heavily enough to be absolutely confident that it is a good decision.

**Make your code compile on all platforms.** Do NOT use a script that converts the code from one platform to another.

**Avoid smart-but-not-wise syndrome.** If you saved a few lines of code by linking against something that itself had ten prerequisites, you just created piles of potential compatibility problems. Sometimes the wise choice is to write your own function for a particular need, instead of using one that someone else wrote. However, there are exceptions as described in the discussion of standards above. This is the narrow definition of “smart but not wise”. The broad definition is everything in this paper, in other words write quality software.

**Bundle libraries.** If you must put forth software the only works with a given version of an open source dependency, include the source code for the dependency in with your code distribution and installation configuration. Some commercial programmers tools allow you to bundle their libraries also, but carefully check the legal wording first. That way, the program will see the correct version of that library regardless of what is on the host operating system.

**Avoid using non back compatible libraries.** The Boost library, tends to be very version sensitive. Thus a program that works on Boost version 1.59 may not work on version 1.56 or 1.60. This seems to be because of the historical role of Boost as a testing ground for proposed new features in the C++ language (which in itself implies Boost should be not be used for writing production software). The Boost developers have begun billing it as an advanced feature library, but until they figure out how to ensure back compatibility (which is doubtful) we recommend avoiding Boost. Many other libraries have the same problem.

**Don't develop on non-standard operating systems.** Ubuntu Linux is notorious for including non-standard things (libraries, python modules, etc.) so that software that works on Ubuntu won't work on other Linux distributions. Do your development on a common, standardized version of Linux, then check whether it works on Ubuntu, because doing it the other way around has a high probability of failure.

**Follow good software development best practices.** Some of these are;

- Use a cross-platform version control system (most modern ones are)
- Use a cross-platform bug reporting system (many are now web based)
- Document the code and usage
- Make common operations easy to control
- Test software thoroughly
- Version your releases
- Include plenty of error trapping in the code. Input information should be heavily error trapped.

**Include a test set to validate correct operation.** A good functional testing procedure validates the local compile of the software, acts as a bunch of example documentation, and cuts down on the number of people who will be contacting you to ask for technical support.

**Test the software installation on many different operating systems.** Virtual machines, containers, and cloud services are a great way to do this.

**Test the installation with different versions of the dependent libraries, modules, etc.** Again, virtual machine, containers, and environment module systems are your friend.

**Test on multiple web browsers.** Best practices in web development include working in multiple browsers, and most web tools are made to work in multiple browsers. If a tool only works in a Microsoft browser, ban it from use in your organization. Cross platform compatibility is one of the biggest benefits of a web browser. Don't throw that away.

**Assume users of the software will do dumb things.** Test the software with many types of invalid inputs.

**Expect network interruptions.** If the software runs over a network, design it to expect network issues like packet loss, and loss of connectivity. Software can be written to continue functioning or pause until the network connectivity is restored.

**Eliminate hard coded paths.** Don't assume that all computers put their libraries, or software, or data in the same directory. Many large installations have a shared file system that may be mounted in some other path. Paths to where your software finds things it needs can be put in the Makefile, a configuration file, environment variables, the input file, a script that launches the program, or command line arguments.

**Use robust algorithms.** There are algorithms that assume that there will be errors in the input data. The most common of these are algorithms that assume that input data will have experimental noise in it.

**Use standard features of the language.** Use compiler flags to only compile ANSI standard code, not proprietary extensions. Occasionally (if not every day) try compiling the software with multiple compiler implementations (i.e. GNU, Intel, Portland Group, Clang. etc.)

**Use cross-platform languages.** Consider using a language designed for cross-platform compatibility, such as Java. Be mindful of the performance penalty compared to a more performant compiled language.

**Graphic interfaces are notoriously non-portable.** It is often best to use a cross-platform language like Java for any software that has a window, drop down menus, etc. Another option is to develop the GUI (graphic user interface) portion to be implemented to run in a web browser, where the development tools and best practices are inherently cross platform. There are also libraries of graphic calls that are specifically designed to run on multiple operating systems.

**Ask the man who owns one.** Before you choose a library or development tool, do a web search on the name of the tool and the word "complaints". See what people who had a bad experience are saying about that tool. Take all of this with a grain of salt, as there will always be a couple fanatics both for and against a product.

**Do not require root or administrator privileges to install or run the software, if at all possible.** In today's security conscious world, it is safe to assume many of the users of your software may not have admin privileges on the computers they use. Tasks that require root or administrator access are often tasks that can seriously break the system if done incorrectly. If you need root access to install your software in a particular directory, it is possible that operating system updates could break that software.

**Security concerns.** Forcing users to use the most recent versions of dependencies only makes sense if it is a network-facing program where security is a big concern. If that is the situation, develop and test for frequent security updates. This often requires even more testing for operating system compatibility. Also, work even harder to minimize the number of dependencies necessary for the software to run.

**Distribute old versions.** Scientific software users often want to be able to choose to standardize each project on the software version it was originally tested on, so they want old versions to be available. In some cases, such as drug development, there may even be a legal mandate placed on your users to keep the original version for some number of years. Distribute old versions of your program for those who have a good business case for using it.

There is an adage in the business community that “You become what you measure”. Find ways to track and test whether your software is meeting your criteria for quality and your software development staff will develop the software accordingly.

## Further Information

A discussion of software robustness
https://en.wikipedia.org/wiki/Robustness_(computer_science)

System quality attributes https://en.wikipedia.org/wiki/List_of_system_quality_attributes

A discussion of software robustness, specifically focused on scientific software
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5390961/

Another discussion that includes the type of insights that come from painful experience.
https://www.backblaze.com/blog/10-rules-for-how-to-write-cross-platform-code/

This book is a good overview, although a number of the software development tools it recommends are a bit outdated.
"Cross-Platform Development in C++: Building Mac OS X, Linux, and Windows Applications", S. Logan, Addison Wesley (2008).

A good synopsis of software testing can be found at
https://www.researchgate.net/publication/273319104_Software_Testing_Overview