Improving Software With Five Lightweight Processes You Can Adopt Now

Phillip Johnston

INTRODUCTION

In a 2018 survey on embedded systems safety and security, Barr Group (2018) found that:

- 38 percent of safety-critical products do not comply with a formal safety standard.
- 43 percent of teams working on safety-critical products do not perform code reviews.
- 41 percent of teams working on safety-critical devices do not perform regression testing (54 percent for Internet of Things (IoT) product teams).
- 33 percent of teams working on safety-critical products do not perform static analysis (49 percent for IoT product teams).

The previous year's results were eye opening as well, as nearly 50 percent of respondents reported not using static analysis, and 36 percent reported that they do not perform code reviews. These numbers are even more alarming given the fact that 25 percent of the reported "internet-connected devices" could kill or injure people if hacked. Twenty-two percent of respondents mentioned that security for connected devices wasn't even on their to-do list. People are becoming increasingly connected, but the standards for safety, testing, and verification are not keeping pace.

I want to be clear: this is unacceptable.

Image: Cecilie_Arcurs via iStock.com

Many teams skip crucial development processes and justify it with scheduling pressure or by blaming the boss. There will always be scheduling pressure, so teams must adjust their approaches, or they will continue to flounder. Bugs are expensive in time, money, and morale. Debugging accounts for 40 to 50 percent of most project costs and schedules (McConnell 1996). Everyone makes mistakes. Anything practitioners can do to keep bugs out of their code or catch them as early as possible will save time and money. The goal is to identify and resolve defects as early as possible. As McConnell points out, the longer a defect remains, the more expensive it is to correct.

I've selected five simple quality improvement processes that teams can adopt over the next month. These processes are cheap or free to implement, apply for any languages and platforms you are using, and best of all–they work. While each process requires a bit of time to get up and running, there is little to no maintenance involved in continuing to use them. These lightweight processes for improving code quality and identifying problems early are:

1. Fix all your warnings.
2. Set up static analysis support for your project.
3. Measure and tackle complexity in your software.
4. Create automated code formatting rules.
5. Have your code reviewed.

PROCESS 1

FIX ALL YOUR WARNINGS

The first thing I do when working on a new project is fix all the compiler warnings. It's amazing how developers will ignore warnings or rationalize their presence. Occasionally, there is even a team that will fight tooth and nail to prevent you from fixing them!

The compiler knows the programming language better than you ever will. You should not ignore the compiler when it is alerting you to an issue. The way you are using the language is dangerous and likely has unintentional side effects. Depending on the warning, you might be introducing undefined behavior into your software (Mertz 2016a; Mertz 2016b).

That's not our idea of quality software.

If you have warnings in your code base, fixing them is one of the fastest ways to improve quality. You will fix bugs and flaws in your program, regardless of whether they are currently problematic.

PROCESS 2

SET UP STATIC ANALYSIS SUPPORT

Static analysis tools provide even better feedback than the compiler. The compiler may happily allow some problematic cases that are legal in language, such as out-of-bounds pointer accesses or missing initialization values. The analyzer will catch these issues and also report red flags such as unused or redundant code. When it is used throughout the development cycle, your static analysis tool will help you catch and prevent latent problems even earlier than your testing cycle.

Some governmental and industrial organizations now require static analysis data for certification processes. Companies such as PRQA provide tools that can check for compliance with safety-critical standards in a variety of industries.

There are many free static analysis tools available, and their commercial counterparts are also relatively inexpensive for companies (most are less than $1,000). My company uses Clang's static analyzer alongside clang-tidy. Endler (2019) provides a curated list of static analysis tools, linters, and code quality checkers for various programming languages, many of which are free.

PROCESS 3

MEASURE AND TACKLE COMPLEXITY

By the time you've eliminated warnings on your project and cleaned up glaring problems exposed by the static analysis tool, you have already made significant progress with software quality. The next goal is to measure complexity in the software. Because highly complex functions tend to be hard to understand, test, and maintain, these functions are prime candidates for refactoring and simplification.

By using a metric to measure complexity, there is a quantitative way to evaluate the code and identify pieces that need special attention. We can see how our changes are impacting the code base over time and trigger automatic alerts and reviews whenever a threshold is exceeded. We can focus our code reviews on functions with high complexity scores, making sure they get the most of our limited attention. Metrics are not perfect, but they increase our insight into software quality.

These are the simplest and most popular metrics for measuring code complexity (Kan 2002):

- **Lines of code (LOC)**: A count of the non-blank, non-comment source lines in a function, module, or project
- **McCabe cyclomatic complexity (MCC)**: Provides a complexity score based on the number of branches (for example, conditional statements)
- **Strict cyclomatic complexity (SCC, CC2)**: Expands MCC by considering the number of conditions within each branch, which provides an approximation for the number of test cases needed for full coverage

Lizard, cccc, SLOCCount, Cmetrics, and CScout are free tools that will calculate complexity metrics for C/C++. My company currently uses Lizard.

PROCESS 4

CREATE AUTOMATED CODE FORMATTING RULES

Automated code formatting might seem like a strange recommendation to put into the top five, but it serves three purposes:

- Automated formatting reduces a programmer's cognitive load by eliminating an entire category of details and decisions they need to keep in mind.
- Automated formatting improves the quality of peer code reviews (the next recommendation) by eliminating arguments about style.
- Automated formatting is the first step toward implementing and enforcing a coding standard.

Every team that I've encountered with a written style guide inevitably ignores those guidelines, and multiple programming styles run rampant. Instead of relying on developers to constantly keep an arbitrary set of rules in mind, we can automate the process to make it simple and impersonal. At the very least, it's worth eliminating the pointless, time-wasting arguments that cause friction within teams. We use clang-format on our projects. Uncrustify and AStyle are other popular code formatting tools.

PROCESS 5

HAVE YOUR CODE REVIEWED

Writers accept the fact that first drafts are generally garbage and need heavy editing. Before I publish articles or newsletters, they have usually gone through

two to three self-editing sessions and one to two peer reviews. Along the way, the text is trimmed and restructured. The result is a much better product than the initial draft.

Yet, for some reason, programmers seem to think perfect code is produced on the first try. The 2018 Barr Group survey results show that 54 percent of IoT product teams don't perform regular code reviews. The survey results also show that a painfully scary 43 percent of teams working on safety-critical software do not perform regular reviews.

Perfect code on the first try might be possible if you're a prodigious programmer. But remember -- even the famed author Ernest Hemingway had an editor. A second set of eyes can identify flaws that were missed in your first pass. Another developer may have different experiences that provide insight into the merits or risks of your approach. The architect on your team probably has input on how a module should interface with other pieces of the program. The "ego effect" also comes into play: knowing that our code will be presented and reviewed by another human can dramatically improve the overall quality (Johnston 2017). We will spend time cleaning up and checking the logic before putting code up for judgment.

Code reviews can waste time and become unproductive if poorly implemented. Johnston (2017) provides some excellent tips for getting started. Notably, a lightweight review process is more efficient and practical than long, in-depth reviews with multiple developers. Even performing reviews on only 20 to 33 percent of the submissions provides benefits due to the "ego effect." While 20 percent may seem low, remember we are aiming for achievable: reviewing 20 percent of the source code is definitely better than none.

I highly recommend implementing peer code reviews after setting up automated code formatting. This helps constrain code review discussions by preventing them from devolving into style nit-picking. If you've set up static analysis, make sure the analysis tools are used prior to code reviews.

THE KEYS TO SUCCESS WHEN ADOPTING NEW PROCESSES

When adopting new processes, it's important to focus on implementing one at a time. Adopting new processes in stages ensures you have time to correctly implement each new technique before moving on to the next one. By implementing too many changes at once, you are likely to overwhelm your team and evoke a mutiny.

If you're leading a team, it helps to find someone who is excited and can help you champion the idea. Empower that person so he or she can demonstrate the benefits of the new process to the team. Back this person up when there is pushback. Change is always hard. Expect resistance, but don't let it stop you.

The key to making these new processes stick is to make them as automated as possible. There is never a case where it isn't worth the time it takes to automate a development process. Automation ensures the process is easy to follow and always happens, rather than trusting individual contributors to remember to follow a process. Automation also makes the process less personal. The rules are clearly defined and are being enforced by a tool. Depersonalization helps us view the situation dispassionately rather than as an attack on our abilities.

Remember, when you implement new processes, adopt just one new process at a time to prevent overloading the team. Next, empower a process champion on your team. Finally, automate!