

The Rules for Writing Maintainable Code

There are three fundamental rules that every software developer should take into account when creating code to support long-term maintainability.

by **Kaushal Amin** | kaushalamin@kms-technology.com

When a developer writes code, he imagines that he will be the only one working on it in the future. But the reality is that someone else will have to work on it. This may be due to a number of reasons: New functionality may be required, changes will be needed for existing features, and fixes for defects will need attention. The latter is a certainty. All of this work is often performed long after the original code was written and by a developer who did not write it. The challenge is to make changes without breaking the existing code. This situation can be complicated by the fact that there may be little technical documentation summarizing what the code actually does, and any future work will typically have tight schedule demands.

If we accept Robert L. Glass's assertion in his post "Frequently Forgotten Fundamental Facts about Software Engineering" for the IEEE Computer Society [1] that software maintenance accounts for 40 to 80 percent of total software development costs, then we can understand the importance of writing maintainable code from the start. Focusing on rushing the product out the door and failing to make code easily understandable for those who will work with it in the future dramatically increases the cost down the line.

Starting over from scratch because you're afraid that everything will break if you make too many changes is hugely disruptive and costly. It's a simple truth that the more maintainable your code is from the start, the longer its lifecycle will be.

The question is, how do you write maintainable code? These three simple rules will keep you firmly on the right track.

1. WRITE CODE THAT IS EASY TO UNDERSTAND AND DEBUG

If the next developer to work on your code can't understand what you've done or why you've done it a *specific* way, then they'll usually throw that code away and start over. It takes longer to understand poorly written code than to write new code from scratch. Write structured code with a clear format, follow conventions, and, if it isn't self-explanatory, make sure

the code is fully commented. Here are a few best practices.

Choose a clear coding style: Keep your function and data naming consistent.

Optimize for the reader, not the writer: Saving time while you write code can cause serious frustration and confusion for anyone reading that code later.

Include concise comments: If it isn't obvious what's happening when you look at the code or you've implemented something a little unusual, make sure you include good comments to explain it. Don't write your comments for yourself—imagine someone else trying to understand your code cold.

Always do the smallest, simplest thing to add value: Always focus on the task at hand and write the best code you can

to achieve your current aim. Don't do anything unusual. Writing code with one eye on future requirements is a recipe for disaster. Designing your code in a modular fashion with separate, discrete parts is much easier to understand.

KISS (Keep It Simple, Stupid): Don't assume the next person working on your code is going to be at the same level of understanding or experience.

Write code that a novice can understand and leave out the experimentation and excessive optimization out.

Ensure good logging of code execution: Effective debugging requires good code logging. You need evidence of what was going on when the code was written. Log actions, entry points, exit points, and parameters, and make the code configurable. When you log from the beginning, it will be easier to pinpoint specific errors and the origins of those errors down the line.

2. WRITE CODE THAT IS EASY TO MODIFY AND ENHANCE

To write code that is truly maintainable, it must be easy to add new functionality and features. Extensibility is vital. If a single change is liable to break the code in ten different places, then you're in serious trouble. It is possible to make your code easier to change down the line. Here's how.

“It's a simple truth that the more maintainable your code is from the start, the longer its lifecycle will be.”

DRY (Don't Repeat Yourself): Many developers have a nasty habit of writing code for one purpose and then copying and pasting elsewhere to do something else. If it gets used in multiple places and there's something wrong with it, then you've just multiplied the defect. If you're tempted to copy and paste code, consider extracting the common functionality to be available throughout your code base.

Separate concerns: You should modularize code based on distinct features that overlap as little as possible in terms of functionality. If the code needs to do fifteen things, then split it up into fifteen modules that each do one thing. Don't try to do all fifteen things in one module because that will make it tougher to make changes without breaking everything else.

Separate code and data: You should always externalize text into separate files. For example, it takes additional effort to isolate menu options and error messages into an external file, but if you put text in the code, it will be more difficult to change it later. Make sure you use a consistent nickname in language text file names. This approach enables text to be updated by nondevelopers without letting them near the actual code.

Avoid long statements and deep nesting: Don't write all your code in one big function because it's really tough to un-

derstand if it's performing too many tasks. In my experience, a single function that is more than a couple of printed pages long is way too long and should be subdivided.

3. WRITE CODE THAT IS EASY TO TEST

Saving the best for last, a good suite of tests can serve as documentation, indicating how the code is supposed to behave while making sure that the code actually supports the expected behavior. Even better, great tests can give you confidence that your code still works after you've made your changes.

Automated unit testing should be implemented from day one so that when you make changes, the automated testing program will run and you can see what needs to be fixed immediately. In agile, even though it takes more time at the outset to write test programs and code concurrently, comprehensive tests should save major time and resources in the long run. **{end}**

Sticky
Notes

Click here to read more at StickyMinds.com.

■ References

index to advertisers

Agile & Better Software Dev Conference West	http://adc-bsc-west.techwell.com	10
ASTQB	http://www.astqb.org/advanced	8
Capgemini	http://www.capgemini.com/testing	25
Cognizant	https://fastest.cognizant.com/webapps/home	Back Cover
HP	http://www.hp.com/go/alm	12
Ranorex	http://www.ranorex.com/whyBSM	2
Software Tester Certification	http://sqetraining.com/certification	1
SQA	http://www.sqasolution.com	13
STARCANADA	http://starcanada.techwell.com	Inside Front Cover
STAREAST	http://stareast.techwell.com	9
Virtusa	http://bit.ly/1ePFwO3	21

Display Advertising
advertisingsales@sqe.com

All Other Inquiries
info@bettersoftware.com

Better Software (ISSN: 1553-1929) is published six times per year: January/February, March/April, May/June, July/August, September/October, and November/December. Back issues may be purchased for \$15 per issue plus shipping (subject to availability). Entire contents © 2014 by Software Quality Engineering (340 Corporate Way, Suite 300, Orange Park, FL 32073), unless otherwise noted on specific articles. The opinions expressed within the articles and contents herein do not necessarily express those of the publisher (Software Quality Engineering). All rights reserved. No material in this publication may be reproduced in any form without permission. Reprints of individual articles available. Call 904.278.0524 for details.