Little Guide on Software Engineering

David Mentré
david.mentre@bentobako.org
Version 0.4

2016-05-21

The purpose of this guide is to give in a concise manner the essential rules of good Software Engineering.

The following rules are given in order, most important first.

Rule 1: Make the right software. From the beginning consult final users. Show them early designs or prototypes and take into account their feedback. The software is developed for them.[2, sec 10, 11 & 45]

Rule 2: Code for humans, not machines. Use meaningful names: your reader should understand the purpose of variables, procedures, methods, packages, ... without having to look at its context or internals.[3, sec 1.1] More generally, make your intent the clearest possible. Develop for others.

Rule 3: Make the software right: test or prove thoroughly your code. Do Unit, Integration, Validation and verification, Resource exhaustion, errors, and recovery, Performance and Usability tests.[2, sec 34 & 43][3, chap 6] Mathematically prove the correctness of the most critical parts, they cannot fail.

Rule 4: Be DRY: Don't Repeat Yourself. Say something only once. If you repeat a piece in a similar way, factorize the common parts. Only have one reference for any piece of data, preferably in text form.[2, sec 7]

Rule 5: Cut program in orthogonal, loosely coupled modules. To manage complexity, use features of your programming language to cut your code in classes, packages, etc. Each module should have a minimal interface revealed to other modules that masks its own "secret". A change in one module should not impact others.[5][4, chap 3]

Rule 6: Document the why in code and the how of interfaces. Use comments to explain the "why" of your code, the non obvious things. The "how" is the code.[2, sec 44] However, for interfaces, it is important to clearly explain your users how to use them, with the most simplest examples.[3, chap 4]

Rule 7: Use version control system and Continuous Integration. Version control systems (git, Subversion, ...) free your mind, allowing to go back at any point in your development an issue would occur.[2, sec 17] By using Continuous Integration, you build, test, prove and release each change. You'll discover issues sooner, you'll be ready at any time.

Rule 8: Specify in advance but use iterative development. Specifications allow to think about the software. Think hard at things you won't change easily later. Define used vocabulary in a dictionary, to share a common understanding. However, whatever plan you'll make, it is going to change. So start with an iterative approach, being prepared to update all software development artifacts.

Rule 9: Automate everything. Don't enter manual commands, use scripts to automatically build, test, prove, deploy, ... your code. Such scripts are reproducible and precisely encode project knowledge.[2, sec 42]

Rule 10: Use contracts and assertions. Contracts and assertions are like executable comments: they document your code but they break if not satisfied and are never out-of-date.[2, sec 21][1, rule 5][4, chap 11]

Rule 11: Use minimal dependencies but don't reinvent the wheel. External dependencies can make your life much easier, you reuse the experience of others. But too many dependencies rapidly become unmanageable, with bugs and security issues you don't understand.

Please comment If you have some feedback, disagree or agree with above rules, don't understand them or have suggestions, let me know.

Acknowledgments Many thanks to Thomas Genet and Yann Régis-Gianas for reviewing draft of this document. Errors are mine.

References

[1] G. J. Holzmann. The power of 10: rules for developing safety-critical code. *Computer*, 39(6):95–99, June 2006.

[2] Andrew Hunt and David Thomas. *The Pragmatic Programmer, from journeyman to master.* Addison Wesley, 1999. ISBN 0-201-61622-X.

[3] Brian W. Kernighan and Rob Pike. *The Practice of Programming.* Addison Wesley, 1999. ISBN 0-201-61586-X.

[4] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition.* Prentice Hall Professional Technical Reference, 1997. ISBN 0-13-629155-4.

[5] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.