

## What's Good Software, Anyway?

**Hakan Erdogmus**

**A** colleague heading a research group recently contacted me. The group had a piece of software that it was about to license out to a company. The software encapsulated the results of several years of work: some fancy algorithms for data manipulation. The company would take over the source code to maintain and evolve it.



My colleague wanted to know what sort of “objectively testable guarantees” he could offer the company about the source code so that the people who will be responsible for its future development “feel comfortable taking on the challenge.” Good question! A challenge indeed.

### **It's not just the functionality**

Put yourself in the shoes of the development team taking on someone else's source code. How can you trust that the endeavor won't turn into your worst nightmare? Sure, the science and technology underlying the software might be terrific and one of a kind, but if you can't crack the code or understand how it's organized, you won't be able to reap the technology's benefits. Not a good prospect, especially if you have just paid a sizeable chunk of money to acquire the technology.

My colleague wisely acknowledged that functionality is only part of the equation

about trusting a piece of code. The other part, of course, is knowing that the code is comprehensible. His email solicited my opinion about some criteria that he had come up with (in italics).

### **Coding standards: The beginning**

*The software should be written according to a coding standard.* This is a good starting point. To begin with, a coding standard improves the source code's appearance and thus its readability. His communication referred mostly to this aspect of a coding standard. But there's much more to coding standards than rules about proper white-spacing and nesting—in fact, most development environments nowadays provide decent formatting capabilities. Other factors are more central. For example, consistent and as-recommended usage of native language constructs comes to mind. Another is adherence to language idioms and micropatterns, including those regarding the use of standard libraries that provide common data structures and utilities.

My colleague suggested that all method names should be self-explanatory, at the expense of long variable names. I agree.

He also inquired: would it make sense to limit the sizes of source code components, such as classes and methods? I'm not so sure here. Besides, although guidelines regarding component sizes could be part of a coding standard, I prefer to discuss them under design quality. So, let's move on to the next criterion.

### STAFF

Senior Lead Editor  
**Dale C. Strok**  
dstrok@computer.org

Group Managing Editor  
**Crystal Shif**

Senior Editors  
**Shani Murray, Dennis Taylor, Linda World**

Assistant Editor  
**Brooke Miner**

Editorial Assistant  
**Molly Mraz**

Publication Coordinator  
**Hilda Carman**  
software@computer.org

Production Editor  
**Jennie Zhu**

Technical Illustrator  
**Alex Torres**

Publisher  
**Angela Burgess**  
aburgess@computer.org

Associate Publisher  
**Dick Price**  
dprice@computer.org

Membership/Circulation Marketing Manager  
**Georgann Carter**

Business Development Manager  
**Sandra Brown**

Senior Production Coordinator  
**Marian Anderson**

### CONTRIBUTING EDITORS

**Robert Glass, Warren Keuffel,  
Keri Schreiner, Joan Taylor**

### CS PUBLICATIONS BOARD

Jon Rokne (chair), Mike Blaha, Angela Burgess,  
Doris Carver, Mark Christensen, David Ebert,  
Frank Ferrante, Phil Laplante, Dick Price,  
Don Shafer, Linda Shafer,  
Steve Tanimoto, Wenping Wang

### MAGAZINE OPERATIONS COMMITTEE

Robert E. Filman (chair), David Albonese, Jean Bacon,  
Arnold (Jay) Bragg, Carl Chang, Kwang-Ting (Tim)  
Cheng, Norman Chonacky, Fred Douglass,  
Hakan Erdogmus, David A. Grier, James Hendler,  
Carl Landwehr, Sethuraman (Panch) Panchanathan,  
Maureen Stone, Roy Want

**Editorial:** All submissions are subject to editing for clarity, style, and space. Unless otherwise stated, bylined articles and departments, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *IEEE Software* does not necessarily constitute endorsement by the IEEE or the IEEE Computer Society.

**To Submit:** Access the IEEE Computer Society's Web-based system, Manuscript Central, at <http://cs-ieee.manuscriptcentral.com/index.html>. Be sure to select the right manuscript type when submitting. Articles must be original and not exceed 5,400 words including figures and tables, which count for 200 words each.

## Is the design decent?

The next step up from coding standards is design quality. *The software's design must be high quality.*

Since my colleague was interested in objective evidence, let's focus on what's measurable. A noble goal, but easier set than achieved.

A variety of source-code metrics capture low-level design attributes, collectively known as internal quality. These metrics attempt to measure the cognitive complexity of the code base, expressed by such proxies as the components' raw sizes, structural or control flow complexity, coherence, and coupling. Research has shown that some of these correlate with maintainability. As far as external quality is concerned, size measures are particularly notable: empirical evidence about their direct association with defect density and fault proneness is compelling. For object-oriented code, several commercial and open source tools exist to compute a variety of code metrics. If you have data that can serve as a benchmark—say, in-house code that you believe is of an acceptable level of maintainability and reliability—you could make comparisons based on such measurements.

Going back to my colleague's question regarding the sensibility of placing caps on class and method sizes, how do you know whether a given value of a code metric is good or bad? Do universally meaningful ranges exist? Sadly, empirical evidence refutes any threshold effects. For example, researchers haven't been able to discover how large such classes or methods can get before incurring a steep maintainability or reliability penalty.

You could also try to gauge whether the code is well refactored. Among other things, well-refactored code is free of replicated segments. Code replication, also known as cloning, is a bad sign. It's known to be correlated negatively with maintainability. You could run a code analysis tool to discover the extent of redundancy in the code base; if redundancy is low, this could give you some confidence in the software's design quality.

In sum, design quality is elusive to assess objectively, but that doesn't mean

you have to throw up your hands and give up. If you use code metrics or clone analysis, however, beware of their limitations. Lack of universal thresholds and proper benchmarks in particular restrict code metrics' practicality for design quality assessment.

## How much documentation?

*Succinct and useful documentation must accompany the software.* Actually, my colleague first suggested something along the lines that the documentation should be extensive. But I'm skeptical about such a stipulation. So, I took the liberty of replacing the not-so-useful qualifier "extensive" with the slightly more useful qualifier "useful." Whether extensive documentation, or any documentation for that matter, is useful to developers depends on its kind, form, traceability, and, most importantly, accuracy.

Let's examine two documentation-related myths.

*Source code—in particular, the interfaces of all public methods—should be extensively commented.* My colleague wanted to know whether this would make sense. As someone who immediately turns off the autocomment features each time I install a new version of my favorite integrated development environment, I'd have to say no. I don't think my aversion to blanket commenting of source code is unfounded. Although autogeneration of documentation from marked-up source code might achieve a certain level of traceability and save time, can it guarantee usefulness? The following segment is obviously silly but should illustrate the point:

```
/*
 * @return transaction id
 */
public Id getTransactionId(...) {
    ...
    /* find transaction's order
    record and return its id */
    order =
        findOrder(transaction);
    return order.getId();
}
```

Good code should be self-explana-

tory. Although self-explanatory code requires skill, adherence to coding standards and sensible, literate coding practices can help a great deal.

More developer-friendly, and useful, alternatives to traditional documentation exist. One is to document the API using executable usage examples. Automated xUnit-style test cases can illustrate the setup of a public method's environment, parameters, and invocation.

If the development environment supports executable, formal assertions, they are an alternative to nonexecutable, textual comments. These are similar to test cases, except they are in-lined like the infamous print statements used during debugging.

Also useful is a roadmap—a get-started document, a technical tutorial of sorts—that walks users through the code base and its API to illustrate how things are organized. I remember seeing this done neatly with a hyperlinked document that actually exercises the API. You can, with some pain and creativity, achieve the effect with a framework such as FIT to keep such documentation traceable and executable.

It's also worthwhile to explain the general architecture and rationale for important, persistent design decisions. I find such documentation useful if it's compact, relies on standard notations, and is annotated with text when warranted. The idea is to communicate what's central and subtle, what you can't easily reengineer or infer from the source code.

*For each major module, a document of at least N pages should clearly describe its design and operation.* A monkey can produce pages of useless documentation (not that it doesn't have better things to do). Documentation decoupled from source code risks becoming outdated before you can utter "detailed design document"—turning a supposed asset to a liability. Perhaps model-driven approaches will one day become practical for the masses, but until then, consider this: good, comprehensible automated tests can sometimes double as documentation. They are always current and traceable to the source code.

Rebecca Wirfs-Brock's column in the last issue ("Driven to ... Discovering

Your Design Values," Jan./Feb. 2007) will give you additional ideas about alternative forms of documentation that different design approaches offer.

Done with documentation? Not so fast.

What about the software's evolutionary trace? That is, baselines of previous versions, requirements in whatever form they were tracked (use cases, scenarios, user stories, or whatever), bug and issue reports, and such. Of course, you used the proper tools to record all this information. You have it at your disposal, and you can browse through it, examine it, or mine it if needed.

### No salvation without regression tests

*A comprehensive suite of regression tests must accompany the software.* How the acquirer of a piece of source code can do without this is unthinkable to me. Here the qualifier *comprehensive* actually means something. My colleague originally inquired about pairing each module and use-case with a black-box test suite. He was referring to acceptance tests. I'd recast his suggestion as having an acceptance test suite that exercises all identified external functionality. Also throw in integration and system tests.

There's more. Can you change the code without your knuckles going white by relying only on acceptance tests? Enter unit tests. Yes, you want them, and you want them to have high code coverage. If unit tests exist, you can assess their coverage with commercial or open source tools. When the code breaks, they might indeed prove to be the new owner's salvation. Their existence is also a sign of testable code. If it's testable to begin with, chances are that you can keep it testable.

**D**o these criteria collectively constitute a definitive, objective trademark of good software? Are they always necessary and sufficient? Of course not. But at least if you're taking over the ownership of a piece of code, they should let you sleep better, I hope! Let me know what you think at hakan.erdogmus@computer.org. ☺

### EDITOR IN CHIEF

#### Hakan Erdogmus

hakan.erdogmus@computer.org

EDITOR IN CHIEF EMERITUS:

Warren Harrison, Portland State University

### ASSOCIATE EDITORS IN CHIEF

**Education and Training:** Don Bagert, Rose-Hulman Inst. of Technology; don.bagert@rose-hulman.edu

**Empirical Results:** Forrest Shull, Fraunhofer Center for Experimental Software Engineering, Maryland; fshull@fc-md.umd.edu

**Design:** Philippe Kruchten, University of British Columbia; kruchten@ieee.org

**Human and Social Aspects:** Helen Sharp, City University, London; h.c.sharp@open.ac.uk

**Management:** Stan Rifkin, Master Systems; sr@master-systems.com

**Processes and Practices:** Frank Maurer, University of Calgary; maurer@cpsc.ucalgary.ca

**Quality:** Annie Combelles, DNV/Q-Labs; annie.combelles@dnv.com

**Requirements:** Roel Wieringa, University of Twente; roelw@cs.utwente.nl

### DEPARTMENT EDITORS

**On Architecture:** Grady Booch, IBM; grady@booch.com

**Bookshelf:** Warren Keuffel, independent consultant; wkeuffel@computer.org

**Design:** Rebecca Wirfs-Brock, Wirfs-Brock Associates; rebecca@wirfs-brock.com

**Glossary:** Richard Thayer, Calif. State Univ. Sacramento; thayer@csus.edu

**Loyal Opposition:** Robert Glass, Computing Trends; rglass@indiana.edu

**Not Just Coding:** J.B. Rainsberger, Diaspar Software Services; me@jbrains.info

**Open Source:** Christof Ebert, Vector Consulting; christof.ebert@vector-consulting.de

**Requirements:** Neil Maiden, City University, London; n.a.m.maiden@city.ac.uk

**Tools of the Trade:** Diomidis Spinellis, Athens Univ. of Economics and Business; dds@aueb.gr

### ADVISORY BOARD

Stephen Mellor, consultant (chair)  
 Jennitta Andrea, ClearStream Consulting  
 Maarten Boasson, Quaerendo Inventietis  
 J. David Blaine, consultant  
 David Dorenbos, Motorola Labs  
 Kaoru Hayashi, SRA  
 Simon Helsen, SAP  
 Juliana Herbert, ESICenter UMSINOS  
 Dehua Ju, ASTI Shanghai  
 Gargi Keeni, Tata Consultancy Services  
 Karen Mackey, Cisco Systems  
 Tomoo Matsubara, Matsubara Consulting  
 Steve McConnell, Construx Software  
 Dorothy McKinney, Lockheed Martin Space Systems  
 Bret Michael, Naval Postgraduate School  
 Susan Mickel, Lockheed Martin  
 Ann Miller, University of Missouri, Rolla  
 Deependra Moitra, Infosys Technologies, India  
 Frances Paulisch, Siemens  
 Suzanne Robertson, Atlantic Systems Guild  
 Grant Rule, Software Measurement Services  
 Wolfgang Strigel, QA Labs  
 Dave Thomas, Bedarra Research Labs  
 Rob Thomsett, The Thomsett Company  
 Laurence Tratt, King's College London  
 Jeffrey Voas, SAIC  
 John Vu, The Boeing Company  
 Simon Wright, SymTech