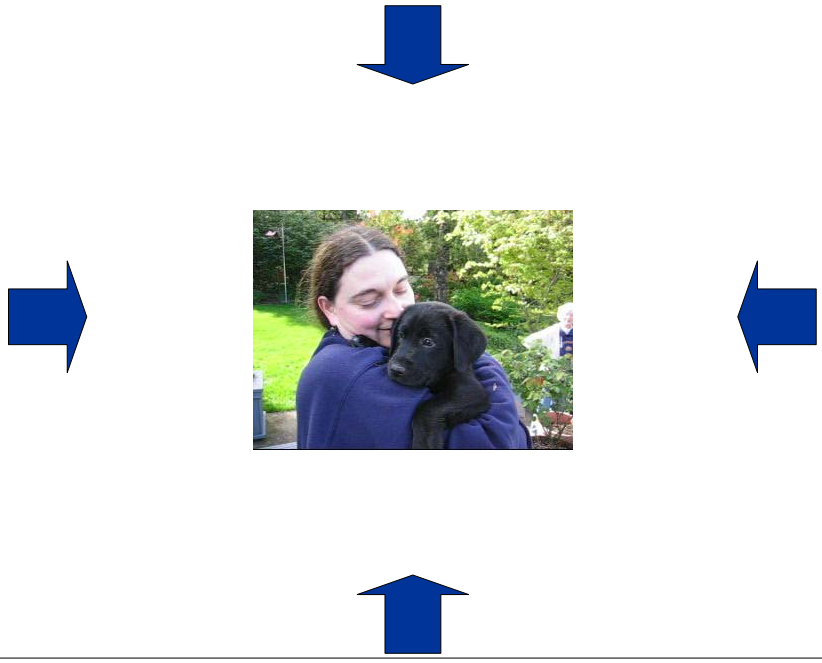
Programmer Discretion and Software Quality

Scott Meyers, Ph.D.
Software Development Consultant

smeyers@aristeia.com
http://www.aristeia.com/

Voice: 503/638-6028
Fax: 503/638-6614

I Come in Peace…



Programmers and Software Quality

"Software quality" is often codespeak for

- **Testing**: attempting to identify defects in the behavior of the software.

Sometimes it also means

- **Requirements**: identifying and communicating what constraints the software should fulfill.
  - ➡ Requirements are often communicated via a *specification*.

In discussions of software quality, *programming* (coding, construction) is rarely mentioned.

Programmers and Software Quality

This is ludicrous.

- **The quality of a software system is strongly affected by the design and coding techniques used to implement the system.**

Specifications and testing are important, but they alone cannot ensure quality:

- **In general, specs will never be detailed enough** to eliminate programmer discretion.
  - ➡ More on this in a moment.
- **In general, testing cannot test everything**.
  - ➡ For black box testing, the set of possible inputs is just too large.
  - ➡ For white box testing, the system may be too large to allow for 100% path coverage — or even 100% statement coverage.

Programmers and Software Quality

Requirements and testing are often less than ideal, anyway:

- Specs are often inaccurate or incomplete.
- Testing is often ad hoc.

Implications:

- In general, we need to do a better job with requirements and testing.
  - ➡ This is one reason why "software quality" often implies "requirements and testing."
- Programmer discretion is even more important than it otherwise would be.

Programmer Discretion

No matter how good the spec, programmers still make lots of little — but important — decisions every day.

Example: The spec says that graphing of the data must be supported.

- Does the graph appear in the app window, in its own window, or is this user-definable?
- Can the graph and/or its window be resized by users?
- Is there a limit on the number of data points used in the graph? On the number of data series in the graph?
- Is the graph editable? If so, does that affect the underlying data?
- Must axes be linear, or can logarithmic, etc., scales be used?
- What about support for color, point "markers," fonts, etc.?
- Is there a command-line or programmatic interface to graphing functionality?

Programmer Discretion

A spec detailed enough to eliminate all such questions would eliminate the need for coding.

- **Unless the spec *is* the program, programmer discretion will affect the behavior — hence the quality — of software systems.**

Two Kinds of Software Quality

Requirements and testing generally address only one aspect of software quality:

- **External quality**: The externally observable characteristics of the software.
  - ➡ Does it do what it's supposed to?
  - ➡ Does it do it as fast as it should?
  - ➡ Is it easy to understand how to make it do it?

"Software quality" people tend to focus on this.

- So do customers.

Two Kinds of Software Quality

Programmers tend to also worry about another kind of quality:

- **Internal quality**: The characteristics of the code itself.
  - ➡ Is it comprehensible?
  - ➡ Is it portable to new platforms (e.g., new compilers, new OSs)?
  - ➡ Is it flexible enough to easily add new capabilities?
  - ➡ Is it robust enough to adapt to new uses or environments?

Some programmers argue that this is the only meaning of "code quality."

- They are mistaken.
- Code that looks good but doesn't work is low-quality code that just happens to look pretty.
  - ➡ Pretty code that behaves badly is no better than pretty people who behave badly.

Customers also care about internal quality, but they may not know it.

- They care when a "simple" change can't be implemented.

Two Kinds of Software Quality

Software of truly high quality needs **both** kinds of quality.

Low external quality leads to:

- Too few users (i.e., customers).
- Low user satisfaction ➡ low user loyalty.
- High support costs (e.g., documentation, tech support, etc.)

Low internal quality leads to:

- Code that can't have new features added.
- Code that won't port to new platforms.
- Code whose bug count refuses to drop ("whack-a-mole" code.)

Two Kinds of Software Quality

There may be tension between external and internal quality:

- Often, higher complexity ➡ lower quality.
  - ➡ More complex systems are harder to write, test, document, maintain, and use.
- Many systems exhibit conservation of complexity.
  - ➡ Complexity can be moved around, but not eliminated.
- In such systems, increasing quality (via complexity reduction) in one place typically decreases quality (via complexity increase) elsewhere.

When that happens:

- Move the complexity to where the fewest people will encounter it.
  - ➡ This usually means simplifying an interface at the expense of an increasingly complicated implementation.
  - ➡ It's an exercise in encapsulation.
    - ◆ Whether it's done is often at the discretion of programmers.

Programmers and Software Quality

Because programmer discretion affects both kinds of quality,

- **You can't improve overall software quality without addressing programmer behavior (i.e., how they exercise their discretion).**

General approach:

- Work to establish a "quality attitude:" an *insistence* on quality.
  - ➡ Start with *insistence*. Compromise only if you have to.
  - ➡ Most people start with compromise.
  - ➡ It shows.
- Recognize that *all* defects are embarrassments worthy of addressing.
  - ➡ If correction isn't practical, prevention may be.
- Provide specific guidelines on how to achieve higher quality.

Beyond Programmers

A "quality attitude" is all-encompassing:

- You either have it or you don't. You can't "sort of" have it.
- *Everybody* has to have it. Quality must be everyone's concern.

Quality is the result of *global* optimization, not local optimization:

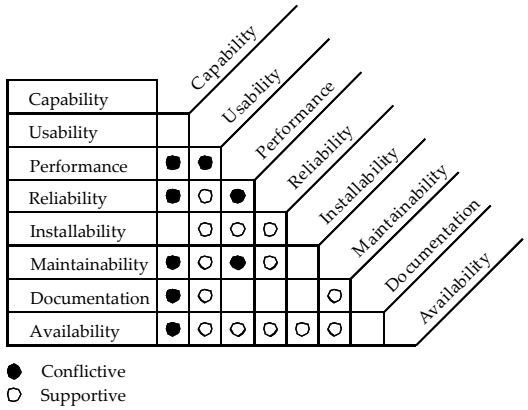
If each person achieves quality on their job, that's no guarantee that the overall result will exhibit quality.

- Example: "It is correct behavior for all auto-numbers to go to 0 when pagination is frozen."
- Example: The team for subsystem A focused on performance, the team for subsystem B focused on portability.
  - ➡ The resulting system was neither "performant" nor portable.
- Example: "I have edited your article in accord with our standards."

Better in What Way?

It's usually impossible to optimize for everything at once.

- From Kan's *Metrics and Models in Software Quality Engineering, 2nd Edition*, pg. 5:



Better in What Way?

- From Spinellis' forthcoming *Code Quality*:

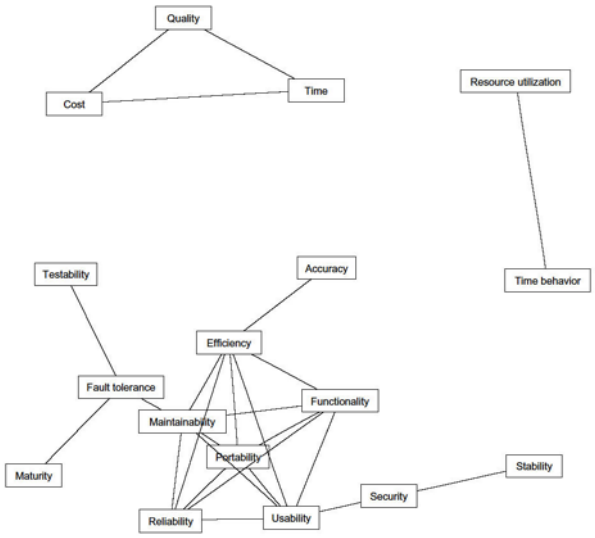


Figure 1.3: Conflicts between quality characteristics.

Controlling Programmer Discretion

Developers thus need guidance on how to exercise their discretion.

- If they know what's most important, they tend to optimize for that.
- If they don't know, each will optimize for what they perceive to be most important.
  - ➡ Different people tend to optimize for different things.
    - This is one reason why multiple-person peer reviews can be so effective.

The following guidelines are from my day-long seminar,
"Better Software – *No Matter What*."

Guidelines for Programmers

Insist on a useful specification.

- Design by Contract (DbC) and assertions can formalize specs and detect violations.

Make interfaces easy to use correctly and hard to use incorrectly.

- Adhere to the principle of least astonishment.
  - ➡ Choose good names.
  - ➡ Shoot for "nice" classes.
  - ➡ Be consistent.
- Introduce new types to prevent common errors.
- Minimize potential resource leaks.

Guidelines for Programmers

Embrace static analysis.

- Compiler warnings
- lint-like tools
- Custom parsers and analyses
- Manual code reviews

Minimize introduction of keyholes.

- Avoid imposing gratuitous restrictions.
  - ➡ Determine limits dynamically instead of statically.
  - ➡ Prefer variables to constants.
  - ➡ By default, view restrictions as gratuitous.

Guidelines for Programmers

Minimize duplication.

- Apply commonality and variability analysis.
- Refactor mercilessly.
  - ➡ Especially after copy-and-paste.
- Consider aspect-oriented programming (AOP).

Consider test-driven development.

Develop iteratively.

Guidelines for Programmers

Remember that you're special — but not that special.

- The guidelines apply to you, too.

Summary

Good requirements and good testing alone can't ensure software quality.

- Programmer discretion is critical.

External quality focuses on program behavior, internal quality on code.

- High quality software requires both.

Quality is the result of global optimization, not local optimization.

- A "quality attitude" should pervade a software development organization.

It's not possible to simultaneously maximize all measures of quality.

- It's important to be clear on which measures of software quality are most important.

Guidelines can help programmers better exercise their discretion.

Suggested Reading

Almost Everything to do with Writing Good Code:

- *Code Complete, Second Edition*, Steve McConnell, Microsoft Press, 2004, ISBN 0-7356-1967-0.

Tensions Among Quality Attributes:

- *Metrics and Models in Software Quality Engineering*, Second Edition, Stephen Kan, Addison-Wesley, 2003, ISBN 0-201-72915-6.
- *Code Quality: The Open-Source Perspective*, Diomidis Spinellis, Addison-Wesley, 2006, ISBN 0-321-16607-8.
  - ➡ Due out March 17.
  - ➡ Information available at http://www.spinellis.gr/codequality/.

Suggested Reading

Design by Contract and Assertions:

- "Building bug-free O-O software: An introduction to Design by Contract," *Eiffel Software web site*, http://archive.eiffel.com/doc/manuals/technology/contract/.
- "An Exception or a Bug?," Miro Samek, *C/C++ Users Journal*, August 2003, http://www.quantum-leaps.com/writings.cuj/samek0308.pdf.
  - ➡ How to use assertions as a form of DbC in embedded applications.
- "A Different Take on Asserts," Jack Ganssle, *Embedded Systems Programming*, December 2003, http://www.ganssle.com/articles/adifasts.htm.
  - ➡ How to use asserts to flag timing failures in real-time systems.
- "Programming with Contracts in C++," Christopher Diggins, *Dr. Dobbs Journal*, March 2005, http://www.ddj.com/documents/ddj0503f/.

Suggested Reading

Interface Design in General:

- "The Most Important Design Guideline?," Scott Meyers, *IEEE Software*, July-August, 2004, http://www.aristeia.com/Papers/IEEE_Software_JulAug_2004.pdf.
  - ➡ Discusses "easy to use correctly, hard to use incorrectly."
- "Programmers are People, Too," Ken Arnold, *Queue*, June 2005.
  - ➡ Discusses API design.

User Interface Design:

- *About Face: The Essentials of User Interface Design*, Alan Cooper, IDG Books, 1995, ISBN 1-56884-322-4.

Suggested Reading

Static Analysis — by Machines:

- "Automated Defect Identification," Kevin Smith, *Dr. Dobb's Journal*, February 2003.
  - ➡ An overview of tools for various forms of static analysis.
- "A Report Generator for PC-Lint," Jon Zyzyck, *Dr. Dobb's Journal*, February 2003.
  - ➡ An approach to applying PC-Lint to programs of 100-200KLOC.
- "Lint Metrics & ALOA," Ralf Holly, *C/C++ Users Journal*, June 2004, http://pera-software.com/aloa/aloa.pdf.
  - ➡ Describes a tool for analyzing output from PC-Lint.
- "Security: The Root of the Problem," Marcus Ranun, *Queue*, June 2004.
  - ➡ How security issues could be addressed via automatic static and dynamic analysis.

Suggested Reading

More Static Analysis — by Machines:

- "Using Redundancies to Find Errors," Yichen Xie and Dawson Engler, *Proceedings of SIGSOFT 2002/FSE-10*, November 2002. Available at http://www.stanford.edu/~engler/p401-xie.pdf.
  - ➡ Empirical report on how suspicious constructs (e.g., assignments to self, dead code, etc.) often indicate more serious coding errors.

Suggested Reading

Static Analysis — by Humans:

- *Peer Reviews in Software*, Karl Wiegers, Addison-Wesley, 2002, ISBN 0-201-73485-0.
  - ➡ Applicable to more than just source code (e.g., requirements documents, design documents, user documentation, etc.).
- *Code Complete, Second Edition*, Steve McConnell, Microsoft Press, 2004, ISBN 0-7356-1967-0, chapter 21, "Collaborative Construction."
  - ➡ A overview of inspection and other review approaches.
- "New Tricks: How Open Source Changed the Way My Team Works," Stephane Lussier, *IEEE Software*, January/February 2004.
  - ➡ Testifies to many beneficial effects of routine code reviews.

Keyholes:

- The Keyhole Problem web site, http://www.aristeia.com/TKP/.
  - ➡ Draft chapters of the book I'm working on.
  - ➡ Information on a mailing list on keyholes.

Suggested Reading

Duplication:

- *Refactoring: Improving the Design of Existing Code*, Martin Fowler, Addison-Wesley, 1999, ISBN 0-201-48567-2.
  - ➡ Introduces the notion of "code smells."
- *The Pragmatic Programmer*, Andrew Hunt and David Thomas, Addison-Wesley, 2000, ISBN 0-201-61622-X.
  - ➡ Popularized *DRY* (Don't Repeat Yourself).
- "Continuous Design," Jim Shore, *IEEE Software*, January/February 2004, http://www.martinfowler.com/ieeeSoftware/continuousDesign.pdf.
  - ➡ Contains anecdotes on how avoiding duplication has allowed systems to be easily enhanced with unforeseen functionality.
- *Multi-Paradigm DESIGN for C++*, James Coplien, Addison-Wesley, 1999, ISBN 0-201-82467-1.
  - ➡ Difficult going, but explains commonality and variability analysis.

Suggested Reading

Aspect-Oriented Programming (AOP):

- "Aspect-Oriented Programming & AspectJ," William Grosso, *Dr. Dobbs Journal*, August 2002.
- The Aspect-Oriented Software Development web site, http://aosd.net/.

Interception:

- "Java Dynamic Proxies: One Step from Aspect-oriented Programming," Lara D'Abreo, July 9, 2004, *Devx.com*, http://www.devx.com/Java/Article/21463.
- "Object Interconnections: CORBA Metaprogramming Mechanisms, Part 1: Portable Interceptors Concepts and Components," Douglas C. Schmidt and Steve Vinoski, *C/C++ Users Journal C++ Experts Forum*, March 2003,http://www.cuj.com/documents/s=8247/cujcexp2103vinoski/.
- "Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse," Dharma Shukla, Simon Fell, and Chris Sells, *MSDN Magazine*, March 2002, http://msdn.microsoft.com/msdnmag/issues/02/03/AOP/default.aspx.

Suggested Reading

Test-Driven Development (TDD):

- "Test Driven Development" wiki page, http://c2.com/cgi/wiki?TestDrivenDevelopment.
- "Test-Driven Development: Concepts, Taxonomy, and Future Direction," David Janzen and Hossein Saiedian, *IEEE Computer*, September 2005.
- "The Test Bus Imperative: Architectures that Support Automated Acceptance Testing," Robert C. Martin, *IEEE Software*, July/August 2005.
- *Test-Driven Development*, Kent Beck, Addison-Wesley, 2003, ISBN 0-321-14653-0.
- http://www.testdriven.com/, a web site devoted to TDD.
- The JUnit web site, http://www.junit.org/.
- *Working Effectively with Legacy Code*, Michael C. Feathers, Prentice Hall, 2005, ISBN 0-13-117705-2.
  - ➔ Lots of information about adding tests to an existing code base.

Suggested Reading

Refactoring:

- *Refactoring*, Martin Fowler, Addison-Wesley, 1999, ISBN 0-201-48567-2.
- *Refactoring to Patterns*, Joshua Kerievsky, Addison-Wesley, 2005, ISBN 0-321-21335-1.

Iterative Development:

- *Extreme Programming Explained*, Kent Beck, Addison-Wesley, 2000, ISBN 0-201-61641-6.
  - ➡ Now out in a second edition.

Please Note

Scott Meyers offers consulting services in all aspects of the design and implementation of software systems. For details, visit his web site:

http://www.aristeia.com/

Scott also offers a mailing list to keep you up to date on his professional publications and activities. Read about the mailing list at:

http://www.aristeia.com/MailingList/