2006

Good/Fast/Cheap: Contexts, Relationships and Professional Responsibility During Software Development

Marty J. Wolf
Bemidji State University

Frances S, Grodzinsky,
Sacred Heart University

Recommended Citation

Wolf, M. J., & Grodzinsky, F. S. (2006). Good/fast/cheap: Contexts, relationships and professional responsibility during software development. In *Proceedings of the 2006 ACM symposium on applied computing* (pp. 261-266). Doi: 10.1145/1141277

Good/Fast/Cheap: Contexts, Relationships and Professional Responsibility During Software Development

M. J. Wolf
Bemidji State University
Bemidji, MN USA

218 755 2825

mjwolf@bemidjistate.edu

F. S. Grodzinsky
Sacred Heart University
Fairfield, CT USA

203 371 7776

GrodzinskyF@sacredheart.edu

ABSTRACT
Engineering requires tradeoffs [23]. When engineering computer applications, software engineers should consider the costs and benefits to humans as an integral part of the software development process. In this paper we focus on reliability, a central aspect of software quality, and the influence of relationships and various software development contexts on the software developer.

Categories and Subject Descriptors
K.7.4 [**The Computing Profession**]: Professional Ethics – *codes of good practice.*

General Terms
Management, Economics, Reliability, Human Factors.

Keywords
Professional responsibility, software development tradeoffs.

1. INTRODUCTION
A software engineer is required to constantly negotiate tradeoffs when developing a computer application, and software reliability concerns are significant during that process. The IEEE defines reliability as "the ability of a system or component to perform its required functions under stated conditions for a specified period of time" [9]. At each step of software development, decisions are made about functionality, the conditions under which the software will be allowed to operate, and how much time and money will be invested in trying to achieve and assess a particular level of reliability.

We contend that competent engineering requires that human values be an explicit and central concern in the process of software development. The people to be considered include the developers themselves, their employers, the people who will buy the software, the people who will eventually use the software, and the "penumbra," people who will be affected by the use of the software. (We use the term "penumbra" to indicate that these people "are under the shadow" of the software; they may or may not even be aware of the software, but it affects them nonetheless [4].) These groups can intersect, but each group represents different collections of interests. In this paper we will focus on three relationships among these groups that we think are most important to the issue of software reliability: the relationships between developers and their employers, between developers and users, and between developers and the penumbra.

SAC'06, April, 23-27, 2006, Dijon, France.

Others have analyzed different influences and incentives that can affect the software development process with respect to software reliability [8] and described legal considerations as well [11]. In this paper we discuss relationships that impact professional responsibilities. As we will see, these professional responsibilities are also greatly influenced by the software development context. Regardless, we insist that professional responsibilities to enhance software reliability extend beyond the letter of the law.

2. HOW RELATIONSHIPS IMPACT SOFTWARE ENGINEERING RELIABILITY
A central reality of software development is that perfection is not an achievable goal [4]. While there may be an occasional opportunity for a piece of software to approach perfection, more often than not, the "good-fast-cheap, pick two" [19] engineering tradeoff is unavoidable. Note that "good" includes many qualities and that reliability is one quality that we understand reasonably (but not perfectly) well. Furthermore, as noted by Lyu, software reliability is a key factor in software quality [14]. Thus, as we discuss "good/fast/cheap" tradeoffs, we will focus on reliability even though the arguments could be made in a similar fashion about other aspects of quality.

Various people have different power in making decisions and different stakes in the results. The developers and their employers are driven by the unique realities of the software market. At least in part because market-share is particularly important in software, software developers are delivering software faster and faster [6]. "Fast" and "cheap" are particularly important for the developer's employer. The "good" of software reliability is in tension with "fast and cheap". In the long run, poor reliability may be a disadvantage for a software provider, but that concern may move

to the background in the rush to deliver a feature-rich application before the release of a competitor's product.

Software users and software buyers want good software; they do not want to pay more than they have to, and they want it as quickly as possible. However, different users emphasize different combinations of good/fast/cheap. Individual users may be more price conscious than a large corporation. Users buying software that is life critical (for use in a pace-maker or aircraft control, for example) will tend to emphasize reliability, and users buying computer games may emphasize "fast", also wanting the latest and greatest features.

The people who may be most affected by the software may be the least aware of the software. Software affects the "penumbra" even though they need not be directly using the software themselves. For example, a medical technician uses software to control an MRI machine, but the MRI machine and its results most directly affect the person getting scanned. The "good" qualities of software are particularly important for the penumbra. The effects of unreliable software may fall on this group even more than others, and the cost and time factors do not affect them as directly.

Although software engineers are aware of these groups and competing interests, they may be less aware of ethical principles that can assist them in developing a reasoned, prudent investigation of the tradeoffs. For example, Kenneth Alpern has articulated an important principle in his article "Moral Responsibility for Engineers": because developers are in a position to cause greater harm because of their technical expertise, they must exercise greater care to avoid doing so [3]. This principle requires an emphasis on "good" over "fast/cheap", and supports reliability. In *A Theory of Justice,* John Rawls states: justice requires special consideration for the least advantaged [20]. In considering the stakeholders in software reliability questions, the penumbra is often the least advantaged.

We should point out that an emphasis of "good" does not require that "fast" or "cheap" be ignored. Software developers have responsibilities to their employers that include developing software efficiently and enhancing the company's profitability. Clearly, "good" can be over-emphasized in a way that is counter-productive; a piece of software that is perpetually in testing will never reach the market, will never deliver functionality to users, and will never deliver benefits to the penumbra. A company that doesn't deliver any functionality (or always delivers after its competition) will go bankrupt. The engineer's dilemma is not about choosing one or two aspects; rather, it is about weighing and balancing. The ethical principles that we discuss here encourage a particular emphasis in that balancing act, not the exclusion of any aspect.

The idea that developers are obliged to consider users and the penumbra in their technical decisions is explicit in ethics codes relevant to software developers. For example, the Software Engineering Code of Ethics and Professional Practice, adopted by the ACM and the IEEE Computer Society, has as its first principle: "Software engineers shall act consistently with the public interest" [22]. The Code goes on to explain that "(t)he ultimate effect of the work should be to the public good". The Association of Information Technology Professionals Code of Ethics has a similar clause that requires that their members acknowledge a professional obligation to society [2]. We contend that this emphasis on the public good requires that the good/fast/cheap-tradeoffs pay special attention to "good," since software quality affects the public most directly. This emphasis on quality includes an emphasis on software reliability.

Deborah Johnson analyzes the relationship between computing professionals and their customers using three models based on where the decision-making primarily resides: agency, paternalistic, and fiduciary [10]. In the agency model, the computing professional does what the customer requests; decision-making resides primarily with the customer. In the paternalistic model, the computing professional decides what the customer needs; decision-making resides primarily with the professional. In the fiduciary model, decisions are made collaboratively; ideally, decision-making is shared between the professional and the customer, each contributing appropriate information and judgment.

We think this analysis of professional relationships is useful in discussing how software reliability fares in the good/fast/cheap-tradeoff. In the following subsections, we examine three particular contexts that have different constraints, constraints that affect the engineering balancing act. The three contexts we will examine are commercial, off-the-shelf software (COTSS); custom-built software (CBS); and open source software (OSS). As we shall see, the context of the software helps establish which of Johnson's computing professional relationship models is most likely to surface as the tradeoffs are decided.

2.1 Context: Commercial Off-The-Shelf Software

Those negotiating good/fast/cheap-tradeoffs for COTSS bring a myriad of concerns to the table, including marketing data, corporate economic data, legal, and development time projections for a particular project. Thus, there are many influences on the ultimate balance among "good/fast/cheap" and the individual software developer may have little control in the ultimate decision of how good the software will be. Quite likely, the decisions about fast and cheap will establish a *maximum* value for the goodness of the software.

The importance of a COTSS developer pursuing raising "good" is influenced by which of the two broad types of COTSS is being developed: stand-alone applications or components that are to be incorporated by developers into larger projects. An important example of stand-along COTSS is shrink-wrapped software that is mass marketed.

A COTSS developer negotiating good/fast/cheap-tradeoffs for a shrink-wrapped application is writing for a different user than a developer writing a component for a larger system. On the one hand, the consumer who buys and uses shrink-wrapped software is likely to be less sophisticated about computing than someone buying a component for integration into a larger system. Ethically, this lack of expertise requires an increased burden of care on the COTSS developer; a less sophisticated user is at a power disadvantage. On the other hand, a COTSS consumer who integrates a component into a larger program does not exhibit this same power disadvantage. However, when a COTSS component is integrated into new applications, the number of people affected by the reliability of the original COTSS component is greatly magnified. This increased distribution of the original work also imposes an ethical burden on the COTSS component developer.

Intended use is an important ethical consideration for software development. It seems sensible that the good/cheap/fast-tradeoffs will be different for software that monitors a nuclear power plant than for software used to play a distributed Internet game. For shrink-wrapped COTSS applications, this type of distinction can be readily apparent during development. However, developers working on COTSS components may have less knowledge about where their component will be used. The same component could be used in a computer game and in a missile guidance system. This ambiguity about the ultimate fate of software being developed is also true for developers in large organizations; they receive specific assignments on projects about which they know very little.

Making responsible good/cheap/fast-tradeoffs with limited information about the eventual use of software is a difficult challenge. A useful approach to this problem is the idea of "informed consent" for software [16]. In order to make informed consent possible with software, the developer must give the COTSS consumer information about the good/cheap/fast-tradeoffs inherent in the COTSS application or component. This information should be understandable by the expected audience (an audience of greater technical sophistication in the case of components than in the case of shrink-wrapped applications) and available before the consumer decides to buy. (Note that detailed information on the *outside* of a shrink-wrapped application's box would fit this requirement, but that information *inside* the box would not.)

While the COTSS company often has a paternalistic relationship with COTSS consumers, the individual developer working for that company may not be in a strong position to influence that relationship. When a developer does not (and sometimes cannot) know who the consumers are, the developer must make decisions without consulting actual users. Although the developer can take seriously the interests of the users and penumbra, COTSS is a context in which decision-making resides primarily with the company, especially in shrink-wrapped standalone applications.

A major ethical concern with paternalistic relationships is that the party making decisions (typically management) may not adequately protect the parties who live with those decisions (the consumers and developers). Since the employer may be focused on "fast/cheap" for obvious reasons, the public good requires that the professional software engineer act responsibly. A particularly poignant contradiction here is that an employer's focus on "fast/cheap" burdens the software engineer with identifying the penumbra and the threats to them—reducing the amount of time available for actual software development. This shift in priorities to the penumbra may disadvantage the developer in the eyes of the company.

2.2 Context: Custom Built Software

In COTSS, a developer is often in a paternalistic relationship with consumers. In custom-built software (CBS), contracts and other agreements are more likely to result in an agency relationship: the developer does what the user requests.

Assuming that the customer is well informed during negotiations for custom-built software, these negotiations afford him/her explicit power in helping to decide the good/fast/cheap-tradeoff. If the contract is fixed cost, the informed customer knows that the developer has an automatic incentive to emphasize "fast/cheap". If instead the customer makes explicit requirements for reliability and other quality characteristics, then the customer can expect the developer to emphasize "good" more consistently.

In some CBS, the ideal of a fiduciary relationship may develop. When the contract negotiations and the subsequent development include close cooperation between developers and customers, trust can become an integral part of the process. In this context, the responsibility for the penumbra falls on both. The customer establishes the maximum good he/she is willing to pay or wait for, while the developer establishes the minimum good he/she can reasonably to deliver.

The methodologies grouped under the phrase "agile methods" are often cited as being aimed at producing active cooperation between users and developers. The "Manifesto for Agile Software Development" advocates "customer collaboration over contract negotiation" and its principles include: "Business people and developers must work together daily throughout the project" [1]. Whether all agile methods approach these ideals is unclear; what is clear is that the Agile Manifesto advocates them. It seems that CBS, rather than COTSS, is more likely to afford a developer the opportunity for the kind of close cooperation envisioned by the Manifesto and a chance to emphasize reliability.

2.3 Context: Open Source Software

Two strong traditions exist within the open source community. One is GNU's "free software definition" [7] and the other is exemplified by the "open source definition" from OSI [18]. For this paper we are interested in software where the source code is available to every user and, thus, we will include both these traditions under the banner of OSS. For other purposes, the distinctions between these different alternatives and their traditional commercial software counterparts can be important, but they are not important to this paper.

COTSS and CBS both establish a strong distinction between developers and consumers. An interesting aspect of OSS is that this distinction can be less pronounced, suggesting that both the good/fast/cheap-tradeoff and the paternalistic/agency/fiduciary-model have to be looked at differently. Since the first users of OSS are often the OSS developers themselves, the professional/client labels seem less appropriate than they were for COTSS and CBS. In spite of these complications, we still think "good/fast/cheap" and "paternalistic/agency/fiduciary" are worth examining for OSS.

Because a developer or group of developers runs a typical OSS project and is responsible for making decisions about the design of the software and the quality of the code that they will accept into the code base, they are ultimately responsible for the penumbra. In addition, their decisions about "good" directly influence the life of the software project. Since the initial users are the developers, they may be more tolerant of glitches and quirks (bugs) than is acceptable for the penumbra. If the reliability is not high enough, the project will likely terminate quickly because without any marketing money behind the project, it will not develop the strong user support it takes to make an OSS project successful. Thus, at least the interests of the users, if not the penumbra, are closely tied to the interests of the developers.

Much OSS is available for free downloads, and even when OSS is sold, it tends to be inexpensive. The initial price of OSS is almost always "cheap" compared to commercial alternatives. Even though the initial cost may be free, there are few assurances about the cost as problems arise. The user of OSS gives up the certainty of a fixed cost with a maintenance cost that is unknown at the outset. In addition, there are those who think that the low initial price of OSS indicates problems with quality. Some contend that the distributed nature of OSS development results in no central entity that can answer maintenance questions quickly and authoritatively; but OSS advocates counter that the distributed nature of OSS quality improvement has an advantage over commercial software because so many people are involved that "all bugs are shallow" [21]. Furthermore, the user/developer community determines the urgency of fixing a particular bug.

From a developer's perspective, the notion of "fast" is not as meaningful in OSS where there is little concern about market pressure, financing and marketing campaigns. There is an opportunity for a project maintainer to forgo releasing the project as an official release (beta-releases are regularly made available as part of the process) until the maintainer is satisfied that the minimum good threshold has been met. Thus, OSS is not scheduled in the same way commercial products are scheduled. The development of OSS is dictated by a large group of developers, making it more difficult for an OSS customer (or participant) to predict or change the pace of the OSS version development. Of course, in OSS you can make your own customized version of a particular OSS application without fear of violating a licensing agreement.

When developers and users of OSS neither get nor give payment, coercive financial self-interest is no longer a major concern. Developers are not "using" consumers to get their money. Users are not trying to negotiate an unfair deal for software. Instead, both developers and consumers in OSS are cooperating freely in the OSS project.

Software reliability in OSS can be a motivation for becoming involved both as a developer (who can contribute to the development effort to increase reliability) and as a user (who seeks a particular good/fast/cheap-tradeoff available in the particular OSS project). The success of most OSS projects is not judged typically by profits (although some companies do make profits with OSS); the success is judged chiefly by how many people use the software. By this measure, some OSS projects are wildly successful.

In the other contexts we have described, software developers and others in the corporate structure share the burden of care for users and the penumbra. In the OSS context, that responsibility falls entirely upon the software developers. As OSS increases its market share, OSS developers will be increasingly obligated to consider their responsibilities to the people who use and are affected by OSS. While it is true that OSS users participate freely, it is not sufficient for OSS developers to point to the low price and claim "you get what you pay for" if the software is unreliable. The ethical principle of consideration of the public good is clear: OSS developers have professional responsibilities, even though they are different from traditional "professionals" in how their work is rewarded. (OSS developers are often working for intellectual challenge and prestige among programming peers.)

OSS developers have in some sense a built-in "informed consent" advantage: by definition, OSS gives users the option of examining the source code of the application. Although the source code may not be understandable to many OSS users, this transparency of code (rare in commercial projects) is a fundamentally open stance that encourages a trust relationship between developers and users. As with Agile Methods, OSS literature advocates a level of cooperation and "community" for OSS participants that is not encouraged or observed in, for example, users of shrink-wrapped commercial applications.

2.4 Conclusions about the Three Contexts

COTSS, CBS, and OSS share some characteristics. No one can make perfect software; developers in all three contexts have to negotiate the good/fast/cheap-tradeoffs inherent in software development. All developers have professional responsibilities to the users and the penumbra. Reliability is an important characteristic in all three contexts. Particularly unreliable software is unwelcome in all contexts, and particularly reliable software is likely to be rewarded.

Our analysis has shown that developers in a COTSS environment have the least autonomy in raising the standard for reliability and that their best opportunities for doing so come early in the development process. When the concerns of the penumbra are raised at the beginning of a project, those concerns have the best chance of being addressed. Indeed, an even better time may be even earlier in the process as the developer is contemplating

employment with the company. Does the company have a good track record of responsibility to the penumbra? Developers in a CBS environment, especially one that employs Agile Methodologies, have the opportunity to regularly renegotiate the good/fast/cheap-tradeoff with the user, with the developer having a possibly equal say in the reliability decisions. Finally, OSS developers have extensive autonomy in identifying what is "good enough" for their project.

Successful software development is also measured differently in these contexts: COTSS developers are often judged on how many units are sold; CBS developers primarily fulfill a contract; and OSS developers are judged by the size of the community involved with their application. The time and effort costs of software development are similar in all three contexts, but they are distributed differently for OSS (a large, distributed network of self-selected contributors) than they are in COTSS and CBS (a relatively small group of employees). The benefits to developers are also different for most developers in OSS (status and satisfaction) than for COTSS and CBS (employment and satisfaction). Users (and buyers) usually seek different things in OSS (lower price and code transparency) and commercial software (better defined organization and guaranteed support).

Advocates and detractors of OSS differ on whether commercial software or OSS delivers more reliability. See [5], [12], [15]. We do not make a judgment here about whose empirical evidence is more compelling. It may be that some users are more comfortable in relying on a cooperating community for reliable software, and other users are more comfortable in relying on someone with whom they have a commercial relationship. However, it is clear from our ethical analysis that the motivations for increasing the emphasis on "good" in the good/cheap/fast-tradeoffs are different in the three contexts.

We find it interesting that the penumbra will not be directly concerned with the distinctions we've made among these three contexts. Indeed, most people affected by software won't be aware of whether it is COTSS, CBS, or OSS. The penumbra will only learn of the software effects, good or bad. Our emphasis on the public good dictates that all software developers, no matter what the context, should keep a concern for the penumbra as a central part of the good/fast/cheap-tradeoff.

3. FURTHER RESEARCH

In this short paper we have covered less than we have ignored with respect to the ethical concerns of software reliability. For further details about ethical incentives for more reliable software, see [8]. For a detailed case history of the causes and effects of a tragic software error in the Therac-25 radiation machine, see [13]. For a more detailed ethical justification for a software developer's responsibility towards the penumbra, see [17]. For a discussion about legal disincentives against unreliable software, see [11].

ACKNOWLEDGMENTS

Thanks to Keith Miller for his thoughtful suggestions regarding this paper. In composing this paper, we drew from material in a previously published work: Grodzinsky, FS, Miller, K, Wolf, MJ, (2005) Influences on and Incentives for Increasing Software Reliability, forthcoming in Journal *of Information, Communication and Ethics in Society.*

4. REFERENCES

[1] Agile Manifesto (http://agilemanifesto.org/, accessed, August 25,2005).

[2] AITP (http://www.aitp.org/organization/about/ethics/ethics.jsp, accessed August 23,2005).

[3] Alpern, Kenneth. (1983) "Moral Responsibility for Engineers," *Business and Professional Ethics Journal*, 2, no. 2, 39-56.

[4] Collins, W.R., Miller, K., Spielman, B. and Wherry, P. (1994) "How good is good enough? An ethical analysis of software construction and use." *Communications of the ACM*, Vol. 37, No. 1 (January 1994), 81-91.

[5] Forrester, J.E. and Miller, B.P. (August 2000) "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing." *Proceedings of the 4th USENIX Windows Systems Symposium*. Seattle, Washington.

[6] Frände, J. (2001) "Product Strategies in Software Companies" TU-91.167 Seminar in Business Strategy and International Business, Helsinki University of Technology, February 1, 2001, http://www.tuta.hut.fi/studies/Courses_and_schedules/Isib/TU-91.167/Old_seminar_papers/frande_johannes.pdf. Accessed January 30, 2004.

[7] Free Software Foundation. Free software definition. (July 22, 2005), http://www.gnu.org/philosophy/free-sw.html. Accessed August 27, 2005.

[8] Grodzinsky, FS, Miller, K, Wolf, MJ, (2005) Influences on and Incentives for Increasing Software Reliability, forthcoming in Journal *of Information, Communication and Ethics in Society.*

[9] Institute of Electrical and Electronics Engineers. (1990) IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY: IEEE.

[10] Johnson, D. (2001) *Computer Ethics*, 3rd edition. New Jersey: Prentice Hall.

[11] Kaner, C. (2003) "Managing the Process: Liability for Defective Documentation." *Proceedings of the 21st Annual International Conference on Documentation*. October, 2003, 192–197.

[12] Lemos, Robert. (2004) "Security research suggests Linux has fewer flaws," CNET News, http://news.com.com/Security+research+suggests+Linux+has+fewer+flaws/2100-1002_3-5489804.html. Accessed December 20, 2004.

[13] Leveson, N. and Turner, C. (1993) "Therac 25---An Investigation of the Therac-25 Accidents," *IEEE Computer*, Vol. 26, No. 7, July 1993, pp. 18-41.

[14] Lyu, Michael R. (1996) *Handbook of Software Engineering Reliability*. New Jersey: McGraw-Hill.

[15] Microsoft Corporation. Get the facts on Windows Server and Linux. (2005), http://www.microsoft.com/windowsserversystem/facts/default.mspx. Accessed August 27, 2005.

[16] Miller, K. Software informed consent: *docete emptorem*, *not caveat emptor*. *Science and Engineering Ethics*, Vol. 4, No. 3 (July 1998), 357-362.

[17] Moor, James H. (1999) "Just Consequentialism," *Ethics and Information Technology*, p. 66-69, Kluwer Academic Publishers, Netherlands.

[18] Open Source Initiative. The open source definition. (2005), http://www.opensource.org/docs/definition.php. Accessed August 27, 2005.

[19] PCGuide.com (http://www.pcguide.com/ref/hdd/perf/raid/whyTradeoffs-c.html, accessed August 23, 2005)

[20] Rawls, John, *A Theory of Justice*, The Belknap Press of Harvard University Press, 1971.

[21] Raymond, E.S. (2001) "The Cathedral and the Bazaar," in *Readings in Cyberethics*, eds. Spinello and Tavani. Sudbury, MA: Jones and Bartlett.

[22] Software Engineering Code Of Ethics And Professional Practice (Version 5.2) as recommended by the IEEE-CS/ACM Joint Task Force on Software Engineering Ethics and Professional Practices (1999) http://seeri.etsu.edu/Codes/TheSECode.htm. Accessed January 15, 2004.

[23] U.S. Dept. of Energy. Engineering Tradeoff Studies. Good Practice Guide: GPG-FM-003 (March 1996), http://www.sc.doe.gov/sc-80/pdf_file/gpg03.pdf, accessed August 23, 2005.

Marty J. Wolf is a professor of Computer Science at Bemidji State University in Bemidji, Minnesota.

Frances S Grodzinsky is a professor of Computer Science and Information Technology at Sacred Heart University in Fairfield Connecticut. She is a visiting scholar at the Research Center on Computing and Society at Southern Connecticut State University, on the Board of INSEIT and is widely published in the field of computer ethics.