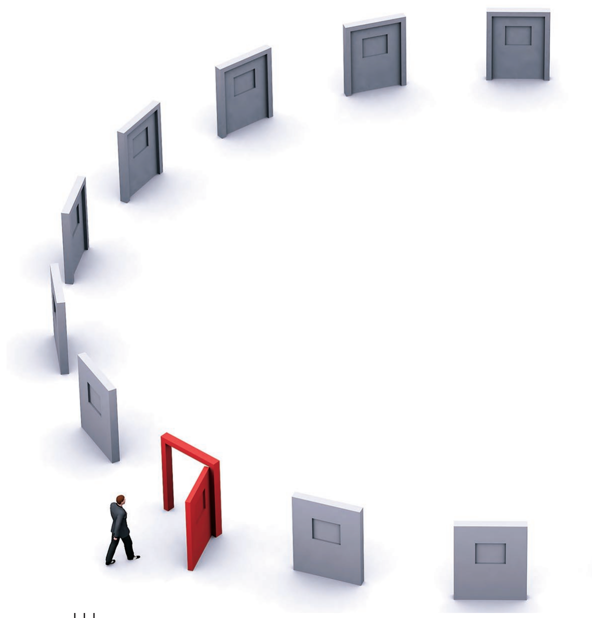

THE PUBLIC IS THE PRIORITY: MAKING DECISIONS USING THE SOFTWARE ENGINEERING CODE OF ETHICS

Donald Gotterbarn, *East Tennessee State University*

Keith W. Miller, *University of Illinois at Springfield*

The Software Engineering Code of Ethics and Professional Practice encourages software engineers to undertake positive actions and to resist pressures to act unethically.

Codes of ethics are often viewed as a way to regulate the behavior of members of a profession. The Software Engineering Code of Ethics and Professional Practice emphasizes self-regulation as well, offering practical advice, fundamental principles, and methods for applying its guidelines in difficult situations.

An important challenge is using the Code to balance multiple factors when deciding on the best course of action. The Code can help a software engineer make complex technical and ethical decisions that are better for the public, the profession, and the engineer. We present three cases—one fictional and two based on news reports—that illustrate how a software professional can use the Code as a decision-making aid when ethical conflicts arise.

CASE 1: DESIGN RISK

Imagine it is 2011, and you work for a company that develops software for military and law enforcement agencies. Your project involves a through-the-wall imaging (TTWI) system that uses impulse radar to see through wood, plaster, concrete, and brick walls (www.defensereview.com/see-through-wall-radar-and-vehicle-disabling-microwave-tech-for-mille-apps). Like other TTWIs, this system uses an impulse synthetic aperture radar system that is capable of remotely imaging targets on the opposite side of the wall at a distance of up to 100 m, with 10-cm accuracy. Your company developed and marketed TTWI, which has become its most profitable product line.

Your new project is to design the software for a stealth disrupter of TTWI signals, called Anti-TTWI. The technical task is not to jam the TTWI signal, but to shift its apparent target by several meters without revealing to the TTWI user that the signal is inaccurate. If the project is successful, your company will be able to sell TTWI to one entity and Anti-TTWI to its adversaries. Your company instructs you not to talk about this project because of national security implications.

You were the lead programmer for the original software for TTWI, and you have the technical skill to develop the

Anti-TTWI. What are your ethical obligations as a software professional in this situation?

THE DEVELOPMENT OF THE CODE

The ACM and the IEEE Computer Society wanted to address both technical and professional issues facing software engineers. To this end, they sponsored the development of a body of knowledge and ethical guidelines documenting the professional responsibilities and obligations of software engineers. A multinational task force including representatives from industry, government, education, and the military compiled a set of guidelines, and 10 years ago, both organizations approved the resulting Software Engineering Code of Ethics and Professional Practice[1] to educate and inspire software engineers. The Code underwent an extensive review process that culminated in the official unanimous approval by the leadership of both professional organizations. It has since been adopted by many other organizations (http://seeri.etsu.edu/se_code_adopter/organizations.asp).

The Code summarizes the software engineering professional's ethical aspirations and explains how these aspirations can affect the way software engineers act. It also informs the public about the responsibilities that are important to this profession and educates practitioners on the standards that society expects them to meet and what their peers strive for and expect of each other.[2]

Principles

The Code includes eight principles and many clauses that detail the application of those principles. The eight principles are arranged with the highest priority—responsibility to the public—appearing first:

1. *Public*—Software engineers shall act consistently with the public interest.
2. *Client and employer*—Software engineers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest.
3. *Product*—Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. *Judgment*—Software engineers shall maintain integrity and independence in their professional judgment.
5. *Management*—Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. *Profession*—Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. *Colleagues*—Software engineers shall be fair to and supportive of their colleagues.
8. *Self*—Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Applying the Code

It is not always easy to wisely apply a collection of abstract principles to concrete actions. To address this problem, the Code differs from many other codes of ethics in two significant ways.

First, many codes are tied to a particular professional organization and apply only to that organization's members. Gehringer's website on computer ethics (http://ethics.csc.ncsu.edu/basics/codes) lists individual codes of ethics for the following organizations: IEEE, Australian Computer Society, Computer Society of India, Hong Kong Computer Society, Association of Information Technology Professionals, Usenix Special Interest Group for System Administrators, National Society of Professional Engineers, ACM, and New Zealand Computer Society. In contrast, the Software Engineering Code of Ethics and Professional Practice is a code of the profession, not a single organization. Several organizations on this list, and many companies not listed here, have adopted the Code as a complement to their own organizational codes.

The Code specifically addresses the problem of conflicting standards, offering techniques to help make ethical decisions.

Second, in addition to providing abstract principles, the Code specifically addresses the problem of conflicting standards, offering techniques to help make ethical decisions. The Code specifically addresses the problem of what to do when standards conflict, as in the Anti-TTWI case. The Code declares that a computer professional should be loyal to his or her employer. It also declares that a software engineer should report any dangers to the public.

In the Anti-TTWI case, these two principles give contradictory advice. On the one hand, the professional's duty to the safety of the public—which includes people targeted by TTWI and those who might be harmed if the Anti-TTWI misdirects weapons—suggests that employees should inform the public about the Anti-TTWI project; on the other hand, loyalty to the employer suggests that the employee keep quiet about this sensitive project. What should a software engineer do when the Code seems to directly contradict itself? The Code makes principle 1—obligation to the public—the priority, resolving this conflict in favor of the public.

The preamble to the Code states:

> Ethical tensions can best be addressed by thoughtful consideration of fundamental principles, rather than blind reliance on detailed regulations. These Principles should influence software engineers to consider broadly who is affected by their work; to examine if they and their colleagues are treating other human beings with due respect; to consider how the public, if reasonably well informed, would view their decisions; to analyze how the least empowered will be affected by their decisions; and to consider whether their acts would be judged worthy of the ideal professional working as a software engineer.

This advice, though helpful, does not make application of the Code to specific situations "automatic." The principles of the Code do not constitute an algorithmic Turing machine that solves ethical problems. Professional judgments are still necessary. The skill of weighing a software engineer's obligations is nontrivial.

The software engineer is obligated to act in the public's best interest, even if those actions oppose the interests of the company.

The Code's treatment of sometimes conflicting ethical principles is not unique. Readers interested in a more detailed philosophical discussion of how practitioners can use different ethical principles harmoniously in computing cases might want to read James Moor's "Just Consequentialism and Computing"[3] or Michael Quinn's *Ethics for the Information Age*.[4]

APPLYING THE CODE TO THE ANTI-TTWI CASE

The Code requires a bias toward the well-being and quality of life of the public: "The Code emphasizes the professional's obligations to the public at large. This obligation is the final arbiter in all decisions … In all these judgments, concern for the health, safety, and welfare of the public is primary; that is, the 'Public Interest' is central to this Code." The primacy of the well-being and quality of life of the public, in all decisions related to software engineering, is emphasized throughout the Code. For example, the whistle-blowing clauses (6.11-6.13) describe obligations for protecting the public when defective software threatens its well-being.

This emphasis on the public good does not remove the software engineer's obligations to the employer in the Anti-TTWI case. Rather, the Code puts the employee's obligations to the employer into perspective. The software engineer should act as much as possible in the interests of the company; however, the primacy of obligations to the public constrains what is ethically permissible. The software engineer must act in a way that enhances the public's safety.

The software engineer has several options in this situation, for example, consulting with managers; seeking a second opinion, perhaps from an ethics advisor or a lawyer; and conferring with executives. But if the software engineer exhausts such options and the company insists on taking actions that compromise the public's safety, then the Code is clear: The software engineer is obligated to act in the public's best interest, even if those actions (at least in the short run) oppose the interests of the company. At that point, the Anti-TTWI case becomes a whistle-blowing case.

The Code provides guidance related to whistle-blowing. A software engineer should

- 6.12. Express concerns to the people involved when significant violations of this Code are detected unless this is impossible, counterproductive, or dangerous.
- 6.13. Report significant violations of this Code to appropriate authorities when it is clear that consultation with people involved in these significant violations is impossible, counterproductive, or dangerous.

Given these directions, the decision is straightforward. The software engineer's first obligation is to discuss the dangers of the Anti-TTWI system within the company and eventually, if necessary, to go outside the company with these concerns. This is not to suggest that these decisions will be easy for the software engineer to make. If, as a matter of conscience, a software engineer becomes a whistle-blower, the personal consequences for the engineer might be catastrophic. While the history of support for engineers who blow the whistle is not particularly encouraging, some relatively recent cases are more promising.[5]

Unfortunately, the ethical challenges that software engineers face are often more complex than the Anti-TTWI case. More complex cases require more subtle ethical judgments. We contend that making such judgments is a technical skill that engineers can learn and practice and that the Code is useful in such learning and practice.

CASE 2: WHO IS IN CONTROL?

On 7 October 2008, a faulty onboard computer suddenly sent a large Qantas passenger jet into a steep dive. The pilot regained control in a few seconds, but meanwhile, 51 passengers and crew were injured, including "broken bones and spinal injuries" (www.abc.net.au/news/stories/2008/10/14/2391134.htm?section=justin). According to that report, "The plane was cruising at 37,000 feet when a fault in the air data inertial reference system caused the autopilot to disconnect." But even with the autopilot off, the plane's flight control computers still command key controls to protect the jet from dangerous conditions, such as stalling, the Australian Transport Safety Bureau (ATSB) said.

"About two minutes after the initial fault, [the air data inertial reference unit] generated very high, random and incorrect values for the aircraft's angle of attack," the ATSB said in a statement. These incorrect values "led to the flight control computers commanding a nose-down aircraft movement, which resulted in the aircraft pitching down to a maximum of about 8.5 degrees."

The software on this Airbus 330-303 implemented a decision to give instant control to the plane's flight control system when the autopilot shut off because of computer system failures. The resulting nosedive suggests that this decision was not in the best interest of the public, especially members of the public in or below this airplane.

There are good reasons to have the flight control system protect the jet from dangerous conditions. But this incident illustrates that the automated decision to turn over control to the flight control system should take into account the current state of inputs into that system. The flight control system should have been more sensitive to the quality of its inputs and to the possibility of disastrous consequences for instantly reacting to apparent conditions that were based on erroneous inputs.

Conflicting requirements

In the Anti-TTWI case, the software engineer faces a difficult situation, but the situation itself is fairly straightforward, even to someone without technical knowledge about the software involved. In the case of the diving aircraft, software engineers made important decisions about how to program the airplane's computers long before the incident occurred, and those decisions involved minute details about how to recognize erroneous inputs and react to different situations that the system might encounter. Software engineers in this type of situation are routinely stretched to the limits of the state of the art in understanding requirements, designing appropriate and safe solutions, and implementing them correctly. This case illustrates that the ethical principle of "public safety first" must be ubiquitous for it to be effective. Only by consistently and diligently applying this principle can engineers hope to avoid situations in which software has injurious or even fatal consequences.

Some might argue that a flight control software problem is not an ethical lapse, but merely a technical problem.[6] We emphatically disagree with that position.[7] We contend that professional ethics are at the heart of this and similar cases. Technical problems are intertwined with ethical nuances, and ignoring either can lead to disaster.

The technical and ethical requirements for the software avionics for this plane are deeply linked. The technical functional requirements provided by the manufacturer address the airplane's structural integrity—the software must not allow the pilots to do anything to damage the airplane. However, it is dangerous if the manufacturer assumes that the plane's environment and computer data will always be correct—and that planes crash primarily because of pilot error. The manufacturer's requirements can be in tension with a pilot's requirement to take over completely when the computer system fails. The system should not prevent effective and necessary human corrective action.

Resolving this situation is not easy. Given the complexity of avionics software, the software engineers in this case must make difficult tradeoffs. But software engineers need to address the ethical problems—largely public safety issues—that these requirements present, and the Code requires that they bring this problem to the attention of those in charge, if necessary. The failure to adequately confront ethical challenges during the requirements phase is evident in many disasters that involve software.

Software engineers are routinely stretched to the limits of the state of the art in understanding requirements, designing appropriate and safe solutions, and implementing them correctly.

The principles of the Code are designed to support software engineers and managers of software engineers who need to take decisive action in a specific case. The professional software engineer cannot always resolve problems in isolation. Often, others must participate to meet the challenges responsibly. A review of the Minimum Safe Altitude Warning System (www.cs.virginia.edu/~jck/publications/greenwell.ress06.pdf) provides an analysis of a similar case.

In many safety-critical software problems, the media often singles out system operators—pilots, nuclear plant control room operators, x-ray room technicians—instead of the software because the software met the manufacturer's specifications. But we contend that manufacturers' specifications can be flawed in ways that a computing professional can, with training, identify as ethically problematic before the system is deployed.

Aeroflot disaster

The Qantas jet incident was not the first, nor the most tragic, involving an Airbus autopilot. According to a report by the Flight Safety Foundation (http://aviation-safety.net/database/record.php?id=19940323-0), problems with

Figure 1. At first, passenger computer use was blamed for the Qantas incident in case 2, but subsequent investigations cast doubt on that idea. Figure reprinted with permission from Rod Emmerson in G. Ansley, "Computer System at Centre of Inquiry into Mid-air Scare," *New Zealand Herald*, 10 Oct. 2008; www.nzherald.co.nz/world/news/article.cfm?c_id=2&objectid=10536760.

transferring control between the autopilot and the human pilots contributed to a 1994 crash of an Airbus 310 flown by Aeroflot. On a flight from Moscow bound for Hong Kong, the pilot brought his daughter and son into the cockpit, letting them put their hands on the controls as the autopilot flew the plane. The report goes on:

> The captain then demonstrated the same features as he did to his daughter and ended by using the NAV submode to bring the aircraft back on course. As the autopilot attempted to level the aircraft at its programmed heading, it came in conflict with the inputs from the control wheel which was blocked in a neutral position. Forces on the control wheel increased to 12-13 kg until the torque limiter activated by disconnecting the autopilot servo from the aileron control linkage. The autopilot remained engaged however. The aircraft then started to bank to the right at 2.5 degrees/second, reaching 45 degrees when the autopilot wasn't able to maintain altitude. The A.310 started buffeting, which caught the attention of the captain who told the copilot to take control while he was trying to regain his seat. The seat of the copilot was fully aft, so it took him an additional 2-3 seconds to get to the control wheel. The bank continued to 90 deg, the aircraft pitched up steeply with +4.8-g accelerations, stalled and entered a spin. Two minutes and six seconds later the aircraft struck the ground.

All 75 onboard were killed. After such a disaster, we would expect the developers of subsequent Airbus autopilot software to be particularly sensitive to issues of control transfer between pilots and autopilots.

In the Aeroflot crash, much of the publicity focused on the judgment of the pilot in inviting his children into the cockpit. While that appears to have been a contributing factor in the tragedy, the autopilot design was at least as significant. However, the media often pays attention to the human factors, which are easier to explain, and might underreport the importance of technological problems. For instance, in the Qantas case, some reports had originally claimed that the cause was interference from passenger electronics, but aviation experts later debunked that claim.[8] As Figure 1 illustrates, attributing the cause of a catastrophe exclusively to human error is easy if there is little surviving evidence about the operator activity during the emergency.

A software professional has extended responsibility to consider the impact of software deployment in particular contexts. Case 2 illustrates a failure to factor user interactions (the pilots) into the software design when those user interactions could have made a positive difference. The software engineers in these cases might have unquestioningly followed the manufacturer's requests; however, true professionals are not guns for hire merely implementing a client's requests—they bring their judgment to the whole task and think beyond the original specifications.

An analog to this situation is when a system requires user input but the software engineer does not provide adequate mechanisms to guarantee safe inputs. This kind of interface design inappropriately shifts the responsibility for safety to the user.

CASE 3: DISCLAIMING RESPONSIBILITY

In August 2000, at the National Oncology Institute of Panama City, medical technicians modified the computerized cancer treatment planning system that calculated radiotherapy treatments. By late March 2001, 28 patients had been overexposed during radiation therapy for colon, prostate, and cervical cancer. The development of patients' symptoms led to the discovery that this modification contributed to 17 deaths and numerous injuries.

As in case 2, investigators immediately placed the responsibility for the problems on users—in this case, medical technicians who had "misused" the treatment-planning software. Initially, the International Atomic Energy Agency (IAEA), part of the United Nations, released a report that identified the cause of the accidents as user error, which caused the computer to miscalculate the radiation dose.

Later investigation painted a different picture. A team of experts at IAEA reviewed and tested all materials related to these incidents ("Investigation of an Accidental

Exposure of Radiotherapy Patients in Panama"; www-pub.iaea.org/MTCD/publications/PDF/Pub1114_scr.pdf). Their report noted that the software manufacturer included a total disclaimer of responsibility for calculations' accuracy:

> [I]t is the responsibility of the user to validate any RESULTS obtained with the system and CAREFULLY check if data, algorithms and settings are meaningful, correct or applicable, PRIOR to using the results as a part of the decision making process to develop, define or document a course of treatment. [Emphasis in original.]

However, the existence of such a disclaimer does not relieve the manufacturer of its responsibilities to those impacted by the system. The investigators found that the user manual did not clearly explain how to enter the data and that the user interface was inadequate. The manual included statements likely to confuse a user, for example: "once the block is nearly finished, strike enter to close the block contour." The adjective "nearly" seems strange in this context. The investigators' report states, "This, in summary, is the information available to the user, placed in different sections of the manual, from which he/she can infer how to enter the data." If users have to infer how to enter the data, something is significantly wrong.

But this was not merely a problem with the user manual. The software design facilitated the "technicians' errors." The system included several different methods of data input. Although only some of these methods included automatic validity testing, all methods produced a plan. Thus, different plans looked identical, even though some had not been checked for safety.

The investigators' report describes the error that proved fatal to some patients. "The staff performed double checks of the data transfer from the prescription and computer output into the patients' treatment charts, but these checks did not include the treatment time calculated by the computer. It was implicitly assumed that the computer output was correct." All the data input was correct, but the technicians did not perform all internal tests of the computer to verify the calculations. In other words, they assumed that the computer program would perform its function.

The medical results for patients were disastrous. "Additional radiation effects will become apparent over the next months and years, and given the radiation doses received, the morbidity and mortality can be expected to increase. Most of the surviving patients already have serious medical problems related mainly to bowel and bladder overexposure. Most of the untoward bowel and bladder effects cannot be remedied." The radiation equipment hardware was not the problem here; investigation by the US Food and Drug Administration (www.fda.gov/cdrh/ocd/panamaradexp.html) indicated that the problem was with the radiation treatment planning software.

Contributing factors

In this case, investigators identified the contributing factors to the overexposure as

- a lack of treatment plan verification at the Panama National Institute of Oncology,
- the method of entering beam block data into the planning software, and
- interpretation of beam block data by the planning software.

The software developers might have taken comfort in the fact that the report lists the first cause as technicians' error, but the technicians' actions were only one of the conditions necessary for these events to occur. Software engineers could have eliminated two of the three factors that combined to create this catastrophe by designing and implementing a more intuitive method for data entry and providing for more consistent data verification.

The Code provides specific details about software practitioners' obligations, and if they ignore those obligations, they are not acting in good faith as professionals.

We do not claim that computing professionals set out to injure patients in case 3; however, their actions contributed significantly to the eventual injuries. The software was capable of performing its required function but failed to consistently check for safe inputs, resulting in unsafe treatment plans. The tragedy is that the software apparently contained algorithms that could have recognized the danger of these plans, but it did not invoke these algorithms consistently.

The Code is relevant to this case because it requires responsibility to those the software affects. It states:

> Ethical tensions can best be addressed by thoughtful consideration of fundamental principles, rather than blind reliance on detailed regulations. These Principles should influence software engineers to consider broadly who is affected by their work.

Are computing professionals who do not act in good faith with the public interest foremost in their work knowingly unethical or just ignorant about how to behave? At least with respect to the effect on the public, it doesn't matter. The Code provides specific details about software practitioners' obligations, and if they ignore those obligations, they are not acting in good faith as professionals. Ignorance of these obligations, either willful or accidental, is not an excuse. The software engineer, the organizations that educated the software engineer, and the software engineers' professional organization share the responsibility for that ignorance.

Therac 25

As with the Qantas case, case 3 is eerily similar to an earlier, well-publicized tragedy that involved software interface problems. The Therac 25, described in great detail by Chuck Huff (http://computingcases.org/case_materials/therac/therac_case_intro.html), was a machine used for radiation therapy. In 1985 and 1986, six patients were killed or seriously injured when given radiation overdoses. Investigations continued until 1987, when manufacturers recalled the machines for major overhauls, including the installation of a hardware safety system that would override the software problems that were identified as the cause of the overdoses.

The Code includes an explicit warning to software engineers against taking on a project for which they are not qualified.

In both the Panama and Therac 25 cases, confusion about data entry led to disastrous overdoses. The software engineers developing the Panama system should have known about the previous and relevant disaster, and that knowledge should have motivated them to take extraordinary care regarding the communication and checking of dosage limits.

Some engineers suggest that when they apply ethics to their technical creations, they are inappropriately limited in deference to philosophers who know little of engineering.[5] But responsible professionals must do more than merely satisfy external functionality. The quality of software engineers' work exists deep inside artifacts, and those artifacts embody the values engineers employ during their development. The fact that engineers can't immediately see all the consequences of their work does not reduce their responsibility to the public.

Engineers could have greatly reduced the risks to the public in both case 2 and case 3 by more carefully considering the software's impact on humans and by heeding the history of relevant disasters involving software controls. Although the public may sometimes "blame the computer" for such problems, software engineers know better. Computers are notoriously bad at doing what we want them to do, but they are very good at doing what we tell them to do. When software is a contributing cause to a disaster, analysts, developers, and managers bear responsibility. Professional competence and diligent protection of the public are required.

THE PUBLIC GOOD IS MORE THAN PHYSICAL SAFETY

All three of these cases involved the potential for fatalities. While that makes the cases dramatic (and we hope memorable), when the Code mentions the "public good," it includes all ways that a software engineer's work can affect society and its citizens, including cases in which lives are not immediately in danger.

For example, in Great Britain, the Child Support Agency outsourced its IT capability to EDS, a company that delivered the CS2 system in 2003. By 2006, CSA's CS2 system developed a backlog of 300,000 cases and more than $5.2 billion in uncollected child support payments. Because of IT failures, an estimated 60 percent of that money will remain uncollected. EDS's computer system also lost the records of some 25 million childhood benefit recipients, overpaid 1.9 million people, and underpaid approximately 700,000 people (www.silicon.com/publicsector/0,3800010403,39160015,00.htm). The failure to deliver current and future support payments caused significant harm. This system clearly had a negative impact on the public interest and on public trust in the software profession.

A competent software engineer understands the difficulty of developing an effective system in such circumstances and informs the customer of these difficulties. The Code is specific about these responsibilities—for example, "Ensure proper and achievable goals and objectives for any project on which they work or propose." Further, the Code includes an explicit warning to software engineers against taking on a project for which they are not qualified: "Ensure that they are qualified for any project on which they work or propose to work by an appropriate combination of education and training, and experience."

Even if a customer or a manager pressures a developer to deliver a system like CS2 before it is ready, the Code encourages a software engineer to resist pressures to act unethically. Developing a system without sufficient attention to the broader context can lead to significant harm, and a software engineer can appeal to the imperatives of the Code to convince others of the danger.

Some have criticized bankers and financiers for their actions that contributed to the worldwide financial problems that surfaced in 2008. Computing professionals have received less attention for their involvement, but ethical lapses involving risk-calculation algorithms might also have been at the heart of the mishandling of risky investment instruments.[9] As computing becomes increasingly intertwined in the details of our lives, the public good depends more and more on software engineers. Software engineers' ethical responsibilities increase as their influence increases.

The Software Engineering Code of Ethics and Professional Practice fulfills several functions. It informs the profession and the public at large about what software engineers consider to be minimally acceptable software engineering practice, even when a nonprofessional practices software engineering. The Code is intended to be inspirational; it encourages software engineers to undertake positive actions and resist pressures to act unethically.

The examples cited here focus on incidents in which software engineers fell short. We do not want to leave the impression that such behavior is the norm, or to ignore the competent and exemplary work that many software engineers accomplish. Unfortunately, good work gains less attention than disasters, both from the public and from ethics scholars. The public judges the software engineering profession in large part by software failures.

In some sense, the exceptions prove the rule with respect to ethical software engineering: The significant losses attributed to incompetence and ethical lapses dramatize the significant gains from competence and ethical actions by software engineers. Reducing the number and severity of incidents is a useful goal for the profession, and we contend that the Code can help the profession work toward that goal. C

Acknowledgment

The authors thank the anonymous reviewers for several important, constructive suggestions for this article.

References

1. D. Gotterbarn, K. Miller, and S. Rogerson, "Computer Society and ACM Approve Software Engineering Code of Ethics," *Computer*, Oct. 1999, pp. 84-88; www.computer.org/portal/cms_docs_computer/computer/content/code-of-ethics.pdf.
2. D. Gotterbarn, "How the New Software Engineering Code of Ethics Affects You," *IEEE Software*, Nov./Dec. 1999, pp. 58-64.
3. J. Moor, "Just Consequentialism and Computing," *Ethics and Information Technology*, Jan. 1998, pp. 61-65.
4. M. Quinn, *Ethics for the Information Age*, 2nd ed., Addison-Wesley, 2006, pp. 398-408.
5. D. Goodin, "Employee Fired for Probing Bad Guys Awarded $4.7M," *The Register*, 16 Feb. 2007; www.theregister.co.uk/2007/02/16/sandia_verdict.
6. J. Steib, "A Critique of Positive Responsibility in Computing," *Science and Eng. Ethics*, vol. 14, no. 2, 2008, pp. 219-233.
7. D. Gotterbarn, "'Once More into the Breach': Professional Responsibility and Computer Ethics," *Science and Eng. Ethics*, vol. 14, no. 2, 2008, pp. 235-239.
8. G. Ansley, "Computer System at Centre of Inquiry into Mid-Air Scare," *New Zealand Herald*, 10 Oct. 2008; www.nzherald.co.nz/world/news/article.cfm?c_id=2&objectid=10536760.
9. G. Hurlburt, K. Miller, and J. Voas, "An Ethical Analysis of Automation, Risk, and the Financial Crises of 2008," *IT Professional*, Jan./Feb. 2009, pp. 14-19.

***Donald Gotterbarn** is professor emeritus in the Department of Computer Science at East Tennessee State University. He is the chair of the ACM's Committee on Professional Ethics and chaired the executive committee that developed the Software Engineering Code of Ethics and Professional Practice. Gotterbarn has been recognized for his leadership in computer ethics by the IEEE Computer Society, by ACM's SIGCAS, and with NSF grants. Contact him at gotterbarn@comcast.net.*

***Keith W. Miller** was recently awarded the Louise Hartman Schewe and Karl Schewe Professorship at the Department of Computer Science at the University of Illinois at Springfield. His research interests include computer ethics, software testing, and computer science education. Contact him at miller.keith@uis.edu.*