THE ETHICS OF SAFETY-CRITICAL SYSTEMS

JONATHAN BOWEN

There are many practices that should be avoided in order to enhance the safety features of software systems.

Plato is dear to me, but dearer still is truth.
—Aristotle (384–322 B.C.)

The use of safety-critical software is rapidly increasing. Computers are used in safety-critical applications at least an order of magnitude more often compared to a decade ago.

It is important to note, however, that software decisions are often based on economic rather than safety considerations. For example, using computers in fly-by-wire aircraft can make significant fuel cost savings since the computer can be programmed to fly the optimal route with little pilot intervention (that is, if all goes well).

Safety implications are more difficult to assess. Some industry professionals claim increased use of software increases the safety of the system. However, since it is so difficult to measure software reliability, this is difficult to justify in practice.

In the aircraft industry a figure of 10^{-9} accidents per hour is often quoted as a goal in standards. Realistically, software testing alone can achieve failure rates nowhere near this figure [3]. Software diversity (*N*-version programming) and software verification and validation can add some extra reliability, but measurements on the effectiveness of such approaches are difficult to obtain. Indeed, relying solely on software seems to be a foolhardy approach given these figures. Despite this, it is sometimes assumed that software is totally reliable since software does not wear out in the same way hardware does. Instead, the errors occur in a much more random and unpredictable manner.

Software is a digital rather than analog artifact. As such, techniques like interpolation and extrapolation, used by many hardware engineers in calculations, do not apply. Changing a single bit in a computer program can have a very unpredictable effect on its operation, or may be completely benign in some circumstances. The complexity of most software is such that it is extremely difficult to reason about its behavior with great certainty.

Given these problems associated with the inclusion of software in safety-critical systems, the deci-

BEATA SZPURA

sion to depend on software to ensure the safety of a system should be taken with great caution. Backup systems designed to operate as simply as possible are recommended.

Seven Deadly Sins

Any scientifically based activity requires a level of responsibility and it is important those involved understand the associated moral questions. Science has developed technology such as nuclear fission and fusion that have the potential for great harm as well as good. When developing safety-critical systems, it is even more critical to consider the ethical questions involved in software development.

Here we examine a number of undesirable practices to avoid in order to help ensure success in a safety-critical project involving software. While these considerations cannot ensure favorable results, they may help to avoid failure, which is all too easy an outcome in software-based projects. A software developer has a wide choice of techniques to apply. Making a decision as to what is cost effective is a difficult part of the engineering process. It is often recommended that a more rigorous approach be used in critical applications (for example, based on formal methods where mathematically-based specification and reasoning is possible. See below). We consider some of the issues and pitfalls in choosing a suitable approach.

1. **Epideictic** (used for effect). Particular techniques and notations should not be applied merely as a means of demonstrating a company's ability. Similarly, they should not be used to satisfy management whim, or because of peer pressure. For example, many firms have adopted object-oriented approaches and programming languages, perhaps without a great amount of thought. Before a particular approach is applied, it should be determined whether it is really necessary.

Potential reasons may be to increase confidence in the system, to satisfy a particular standard required by procurers, or to aid in tackling complexity.

However, there are occasions when some techniques such as formal methods are not only desirable, but positively required. A number of standards bodies have not only used formal specification languages in making their own standards unambiguous, but have strongly recommended and may someday mandate the use of formal methods in certain classes of applications [2].

2. **Hyperbole** (exaggeration). Sometimes exaggerated claims for the benefits of certain methods are made, but no particular approach should ever be seen as a panacea. For example, a formal method could be just one of a number of techniques that, when applied judiciously, may result in a system of high integrity. Complementary methods should not be dismissed lightly. Despite the mathematical basis of formal methods, they are no guarantees of correctness [5]; they are applied by humans, with all the potential for error this brings.

Tools may be available to support a particular method, but again, the vendor may overstate their value. Most methods depend on the quality and suitability of the personnel involved in applying the techniques since system development is essentially a human activity.

3. **Pistic** (too trusting). Some place too much faith in techniques and tools instead of the reasoning power of humans. A tool may return an answer (possibly true or false), but this depends on both the quality of the input to the tool and on the correctness of the tool itself.

Similarly, no calculation or proof should be taken as definitive. Hand proofs are notorious for not only introducing errors at each proof step, but also for

---

Formal Methods and Formal Specification

Formal methods [5] are one of a number of techniques advocated in the development of safety-critical and other high-integrity systems [3]. These are mathematically-based techniques designed to aid in the correct specification and development of software and hardware systems. Their use typically helps to eliminate errors early in the design life cycle (for example, at the requirements and specification stages) when it is still relatively cheap to do so. Many errors in practice are introduced at this point and are expensive to eliminate later (for example, at the programming or testing stage).

Formal specification concentrates on providing a mathematically unambiguous description of the "what" for a system whereas a program (also a formal representation of the software) must also consider the "how" and is, hence, inherently more complex in general.

Further general information on formal methods, including suggestions of introductory articles and information on specific techniques (for example, the Z notation for formal specification and the B-Method for tool-supported formal development), is available as part of the Web's Virtual Library: archive.comlab.ox.ac.uk/safety.html. **C**

making assumptions that may be unfounded. From experience of machine-checking proofs, most theorems are correct, but most hand proofs are wrong. Even the use of a proof checker does not guarantee correctness. While it does aid in highlighting unsubstantiated reasoning, and avoidable errors, proof should never be considered an alternative to testing, although it may reduce the amount of testing required.

4. **Oligarchy** (rule by a few). Many communities associated with a particular software engineering approach tend to be rather introspective, with a few experts preaching their particular wares. However, there has been considerable investment in existing software development techniques and it would be foolhardy to replace these en masse with any new method. Instead it is desirable to integrate new methods into the design process in a cost-effective manner.

Technology transfer from the oligarchy of experts to the potential users is an extremely important and time-consuming process, critical to the success of any new technology. In the early application of a new method, it is often helpful to have an expert on call or, even better, working directly with the team involved. Ready access to education and training are important aspects of the transfer process. University education allows long-term inculcation of ideas and attitudes, while training courses allow engineers to keep up-to-date with new developments [12].

5. **Ephemeral** (transitory). Some techniques are short-lived and others are deemed more useful in the longer term. It is easy to be bamboozled into using a new technique that may not be helpful for the future. It is especially important for safety-critical systems to ensure any techniques used are well established and produce reliable results that improve the overall development process.

Designing with reuse in mind is important for long-term cost benefits, but can have serious safety implications if not done with care. An extremely expensive example of the potential reuse problems is provided by the Ariane 5 disaster, where the inclusion of existing software in a new product resulted in catastrophe [9]. Here the old code was not sufficiently well tested in the new environment since it had always worked in the original environment without problems.

6. **Epexegesis** (additional words). Documentation is a very important aid to understanding, but should not include unnecessary detail. Too much documentation, obscuring the real information content, is as bad as too little documentation. Abstraction can be used to avoid verbosity, but arriving at the right level of abstraction is something that most people find difficult, at least initially, and only comes with experience.

If formal specifications are included, these should always be accompanied by informal explanation to clarify the meaning and place it in context. Formal specifications can be made intelligible and acceptable to nontechnical personnel [5], but only provided they are augmented with sufficient amounts of informal explanation, diagrams, and so on.

7. **Maiandros** (meandering). Taking too long in the development of software is a widespread phenomenon. Estimating the time for software projects is difficult since even given a specification, the complexity of the software to implement it is very hard to assess in any reliable and scientific way.

In a typical project, natural language and diagrams are used to specify the desired system. These delay the use of formalism until the programming stage. Many errors are discovered only if some sort of formalism is introduced, be it a mathematically-based notation for specification or a programming language for implementation.

A specification is typically an order of magnitude smaller than its matching implementation, thus it is much cheaper to produce. It also makes the matching program design easier, since many issues that may have been discovered later have already been resolved. Therefore, the early use of formalism for specification allows the discovery of many errors more quickly and, thus, also at a reduced overall cost.

> *But, my dearest Agathon, it is truth which you cannot contradict; you can without any difficulty contradict Socrates.*
> —Plato (429–347 B.C.)

Ethical Considerations

The origins of western philosophy, ethics, and scientific thinking can be traced back to the Greeks. Indeed, it has been said by John Burnet in his 1892 book on early Greek philosophy that it is an adequate description of science to say "thinking about the world in the Greek way."

Ancient Greece provided the catalyst for much early advancement of knowledge. Socrates (469–399 B.C.), then Plato (429–347 B.C.), and Aristotle (384–322 B.C.) developed philosophical research in a manner far advanced for their time. This was made possible by the Greek way of life, which left time to ponder more abstract questions (at least for a proportion of the population). Their influence is still very much with us today, and their ideas are still under active discussion in philosophical research circles.

Aristotle was a great teacher and left many notes.

Among these were his lectures on ethics, developed over the course of several years and delivered at his Lyceum, recently rediscovered in Athens during excavations for a new building. While these notes were never finished in the form of a complete book, they are of sufficient interest to be the subject of continued study, and to have been reworked into book form by recent researchers [1].

Aristotle's *Nicomachean Ethics* has had a profound and lasting effect on western philosophical thinking since its origins in the 4th century B.C. It has been both supported and rejected, but never ignored. It is inevitably flawed by its very nature, being working notes for lectures. It starts with the object of life (aiming at some good), and proceeds via morals, justice, intellectual virtues, pleasure, friendship to happiness, the ultimate goal, even if it is difficult to define. In this article, we concentrate on Book VI concerning *Intellectual Virtues* [1] in discussing ethics in the context of safety-critical systems.

What is the Right Principle that should Regulate Conduct?

Ethical considerations are a source of much debate, in general as well as with respect to computing applications. Utilitarianism considers the consequences of action and deontological theories, in response to problems with utilitarianism, consider the actions themselves. Social contract theories consider individuals acting as rational free agents constrained by social rules. There is no general agreement about the best approach, but the maximization of happiness is often an aim. Of course, there is individual happiness and overall happiness, which can be in conflict. Risk of death and injury generally result in a decrease of happiness, but sometimes risks such as war may be considered a necessary evil.

The development of a safety-critical system should aim to avoid the loss of human life or serious injury by reducing the risks involved to an acceptable level. This is normally considered an overriding factor. The system should always aim to be safe, even if this adversely affects its availability. It is the responsibility of the software engineering team and the management of the company involved to ensure that suitable mechanisms are in place and are used appropriately to achieve this goal for the lifetime of the product.

It is sensible to follow certain guidelines when developing any software-based artifact [4], especially if there are safety concerns. Most professional bodies such as the ACM, IEEE, British Computer Society, and Institution of Electrical Engineers provide recommended codes for members. Textbooks dealing

---

Code of Practice for Engineers and Managers

Engineers and managers working on safety-related systems should:

• Take all reasonable care to ensure their work and the consequences of their work cause no unacceptable risk to safety;
• Not make claims for their work that are untrue, or misleading, or are not supported by a line of reasoning that is recognized in the particular field of application;
• Accept personal responsibility for all work done by them or under their supervision or direction;
• Take all reasonable steps to maintain and develop their competence by attention to new developments in science and engineering relevant to their field of activity; and encourage others working under their supervision to do the same;
• Declare their limitations if they do not believe themselves to be competent to undertake certain tasks, and declare such limitations should they become apparent after a task has begun;
• Take all reasonable steps to make their own managers, and those to whom they have a duty of care, aware of risks they identify; and make anyone overruling or neglecting their professional advice formally aware of the consequent risks;
• Take all reasonable steps to ensure that those working under their supervision or direction are competent; that they are made aware of their own responsibilities; and they accept personal responsibility for work delegated to them.

Anyone responsible for human resource allocation should:

• Take all reasonable care to ensure that allocated staff will be competent for the tasks they will be assigned;
• Ensure human resources are adequate to undertake the planned tasks;
• Ensure sufficient resources are available to provide the level of diversity appropriate to the intended integrity. **c**

Source: Produced by [6].

It is important to note that software decisions are often based on economic rather than safety considerations.

with ethical issues in computing normally cover at least some of these. Most codes coming from an engineering background place a very high priority on safety aspects, whereas codes of conduct from a computer science background tend to place slightly less emphasis on safety and consider other losses as well, although safety is still an important factor.

In the U.K., there is guidance specifically for engineers and managers working on safety-related systems (see Box 2, extracted from [6]). This code of practice is intended for use by professional engineering institutions such as the IEE, which may wish to augment the code further with sector-specific guidance on particular techniques [11].

There are two types of thought process, the irrational and the rational. Obviously, the professional engineer should aim to use rational lines of thought in the reasoning about and development of a safety-critical system. Unfortunately, development can depend on the personal, possibly unfounded, preferences of the personnel involved, especially those in a management role.

Within the rational part of the mind, Aristotle makes a further distinction. In science, theory and practice are two very important aspects, each of which can help to confirm and support the other. Without a firm theoretical basis, practical applications can easily flounder; and without practical applications, theoretical ideas can be worthless. To quote from Christopher Strachey, progenator and leader of the Programming Research Group at Oxford University:

"It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and hardware design, is unsound and clumsy because the people who do it do not have any clear understanding of the fundamental principles underlying their work. Most abstract mathematical and theoretical work is sterile because it has no point of contact with real computing."

It is highly desirable to ensure that the separation between theory and practice is minimized. This is cer-

tainly extremely important in the area of safety-critical systems. A good theoretical and mathematical underpinning is essential to ensure the maximum understanding of the system being designed. If this understanding is not achieved, serious problems can easily ensue [10].

Five Modes of Thought by which Truth is Reached

The background and expertise of the engineers involved is an extremely important contributing factor to the success of a project. Here we consider five critical aspects of the personnel involved.

Science or scientific knowledge. It is important to have the basic theoretical groundwork on which to base practical application. This is typically achieved through initial education, topped off with specialist training courses. Modern comprehensive software engineering textbooks and reference books normally include a section or chapter on safety-critical systems. Subsequent professional development is also crucial to keep up to date with changes in the state-of-the-art.

There are now a number of courses available specifically for safety-critical systems, especially at the Master's level. For example, the University of York runs a modular Master of Science in Safety Critical Systems Engineering [12] and Diploma/Certificate. The timetable consists of intensive week-long modules—a format found to be much more convenient for industry professionals. The course content is agreed and monitored by a panel of experts, many from industry, to ensure the relevance of the material. The course includes a significant mathematical content, especially in the areas of probability and discrete mathematics for software development.

Art or technical skill. Once the educational groundwork has been covered, greater expertise is gained in the actual application of the techniques learned. For some, applying mathematical skills can be even harder than acquiring them in the first place.

One of the most difficult aspects is learning to model reality with sufficient accuracy. Abstraction is a skill that only comes with practice. Unfortunately, many existing programmers have to unlearn their nat-

> # Every engineer has limitations, and it is important that individuals recognize their own limitations, as well as those of others, and keep within their bounds of expertise.

ural tendency to become bogged down in implementation detail when considering the system as a whole unit. Instead, only the pertinent aspects should be included at any given level of abstraction.

Prudence or practical wisdom. New techniques should be used with great caution in safety-critical applications and introduced gradually. It is best for new approaches to be used on nonsafety-critical systems first to gain confidence, even if they look promising theoretically and are understood by the development team.

Once sufficient experience has been gained, and the benefits have been assessed, then a technique may be recommended and adopted for use in safety-critical areas. For this reason, any prescriptive recommendations in standards should be updated regularly to ensure they remain as up to date as possible.

Every engineer has limitations, and it is important that individuals recognize their own limitations, as well as those of others, and keep within their bounds of expertise. However competent a person is, there will always be tasks that cannot reasonably be tackled. If the constraints on the development of a safety-critical application are impossible to meet, there should be mechanisms to allow this to be expressed at all levels in a company.

A well-known example where such mechanisms were not effective is the Therac-25 radiation therapy machine where several fatal radiation overdoses occurred due to an obscure malfunction of the equipment, the first of its type produced by the company involved to include software [8]. It should be remembered that software by itself is not unsafe; it is the combination with hardware and its environment to produce an entire system that raises safety issues [6].

> *I have hardly ever known a mathematician who is capable of reasoning.*
>
> —Plato (429–347 B.C.)

Intelligence or intuition. The highest quality of personnel should be employed in the development of safety-critical applications. The engineers involved should be capable of absorbing the required toolkit of knowledge and also accurately comprehending the required operation of computer-based systems.

Specialist techniques, including the use of mathematics, are important in producing systems of the highest integrity. Formal specification helps reduce errors at low cost, or even a saving in overall cost. Formal verification, while expensive, can reduce errors even further, and may be deemed cost effective if the price of failure is extremely high (for example, involving loss of life). For the ultimate confidence, machine support should be used to mechanize the proof of correctness of the software implementation with respect to its formal specification. For example, the B-Tool has been used to support the B-Method in a number of safety-critical applications, especially in the railway industry.

For mathematicians, proofs are a social process, taking years, decades, or even centuries to be generally accepted. The automation of proof is regarded with suspicion by many traditional mathematicians. A critical aspect of proofs is seen by many to be their inspectability, leading to understanding of why a proof is valid, and explaining why the theorem under consideration is true.

An automated proof may have millions of (very simple) steps, and be extremely difficult to follow as a result. In addition, any proofs undertaken in the software engineering world must be performed and accepted several orders of magnitude faster than in traditional mathematics (typically in weeks or months rather than years or even decades). However, they are far shallower than most proofs of interest to professional mathematicians. Hence, intuition about why a program is correct with respect to its specification is difficult to obtain in all but the simplest of cases. Unfortunately, this problem will not disappear easily and much research remains to be done.

Collaborations such as the European Provably Correct Systems (ProCoS) projects attempted to explore formal techniques for relating a formal requirements specification through various levels of design, programming, compilation, and even into hardware using transformations based on algebraic laws. Unifying theories for these different levels are needed to ensure consistency. However, it is often difficult to guarantee compatibility of the models used at the various levels

without unduly restricting flexibility.

Wisdom. With experience comes wisdom. It is safest to stick to very modest ambitions if possible to ensure success. Unfortunately software tends to encourage complexity because of its flexibility. This should be resisted in safety-critical applications. The safety-critical aspects of the software should be disentangled from less critical parts if possible. Then more effort can be expended on ensuring the correctness of the safety-critical parts of the system.

> *The unexamined life is not worth living.*
> —Socrates (469–399 B.C.)

Epilogue

The programming profession has traditionally had many of the attributes of a craftsman, such as artistry, but has often lacked foundational knowledge, such as mathematical underpinning of the concepts involved. This is not so important for programs that are not critical for the operation of a system, where errors will not produce disastrous results. However, if safety is of prime importance, such an approach is no longer practical, and in fact is downright unethical.

Personnel involved in safety-critical application development should possess a balance of high-quality skills with regard to all the aspects outlined in this article. Those that cannot demonstrate the appropriate mix of expertise should be restricted to noncritical applications.

No special qualifications are currently required for personnel developing or maintaining software for safety-critical systems. This contrasts with the more established engineering professions where standards, regulations, certification, and accreditation often apply much more strictly.

Some of the words in the section headings in this article may be unfamiliar to some readers. The same problem occurs when new methods are used, especially ones employing mathematical symbols such as formal methods. In the past, the necessary grounding for the use of sound techniques has not been taught adequately in computer science courses, resulting in many software engineers, and even more managers, shying away from their use because they are out of their depth. Fortunately, many undergraduate courses (at least in Europe, and particularly in the U.K.), now teach the requisite foundations to software engineering. Moreover, courses normally give a grounding in ethical issues, often at the behest of professional bodies who may not accredit the course otherwise.

However, the teaching of both ethical and formal aspects is often not integrated with the rest of the syllabus. More coordinated courses in which the mathematical foundations are subsequently applied to practical problems will help to produce more professional software engineers for the future.

Ultimately, it is unethical to develop software for safety-related systems without following the best practice available. All software engineers and managers wishing to produce safety-critical systems in a professional manner should ensure they have the right training and skills for the task. They should be able to speak out without fear of undue repercussions if they feel a system is impossible or dangerous to develop. It is important that companies, universities, standards bodies, professional institutions, governments, and all those with an interest in the well-being of society at large ensure that appropriate mechanisms are in place to help achieve this aim.

> *Eureka!*
> —Archimedes (287–212 B.C.) **C**

REFERENCES
1. Aristotle. *Ethics.* Penguin Books, London, 1976.
2. Bowen, J.P. and Hinchey, M.G. High-Integrity Systems Specification and Design. FACIT series. Springer–Verlag, London, 1999.
3. Bowen, J.P. and Stavridou, V. Safety-critical systems, formal methods and standards. *IEE/BCS Softw. Eng. J. 8*, 4 (Jul. 1993), 189–209.
4. Gotterbarn, D., Miller, K. and Rogerson, S. Software engineering code of ethics is approved. *Commun. ACM 42,* 10 (Oct. 1999), 102–107.
5. Hall, J.A. Seven myths of formal methods. *IEEE Softw. 7,* 5 (Sept. 1990), 11–19.
6. The Hazards Forum. Safety-related systems: Guidance for engineers. The Hazards Forum (1995). London, U.K.; www.iee.org.uk/PAB/SCS/hazpub.htm.
7. Leveson, N.G. *Safeware: System Safety and Computers.* Addison-Wesley, Reading, PA, 1995.
8. Leveson, N.G. and Turner, C.S. An investigation of the Therac-25 accidents. *IEEE Computer 26*, 7 (Jul. 1993), 18–41.
9. Lyons, J. L. ARIANE 5: Flight 501 failure. Report by the Inquiry Board, European Space Agency. (July 19, 1996); www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html
10. Neumann, P.G. *Computer Related Risks.* Addison-Wesley, Reading, PA, 1995.
11. Thomas, M. Formal methods and their role in developing safe systems. *High Integrity Syst. 1*, 5 (1996), 447–451; www.iee.org.uk/PAB/SCS/wrkshop1.htm.
12. The University of York. Safety critical systems engineering, system safety engineering: Modular MSc, diploma, certificate, short courses. 1999. The University of York, Heslington, U.K.; www.cs.york.ac.uk/MSc/SCSE/

JONATHAN P. BOWEN (bowen@sbu.ac.uk) is a computer science professor at the School of Computing, Information Systems and Mathematics, South Bank University, London, U.K., www.jpbowen.com