When Bad Code Comes From Good Specs

Clark Savage Turner, J.D., Ph.D.,
J. Kris Fox
Department of Computer Science, Cal Poly State University
San Luis Obispo, CA. 93407 USA
(805) 756 6133
csturner@calpoly.edu, jfox@calpoly.edu
www.csc.calpoly.edu/~csturner

ABSTRACT

Software code that fails to properly implement a well defined specification involving safety results in executable code that is "more dangerous than it was designed to be." This paper deals only with mistakes programmers make in implementing the design safety specifications. It is shown that such defects in code, if they are a substantial cause of personal injury, result in liability without fault – that is, "strict liability." License disclaimers are ineffective in such cases and the best process in the world is no legal defense. Suggestions are given to lower the risk of liability for such defects in safety-critical code.

KEYWORDS: software products liability, defect, specification

1. INTRODUCTION

Homo Faber, "man, the maker." We like to build things, new things that have not existed before. Things that have an impact in the world. Product developers use science, experience and intuition to create new artifacts with desired properties. Aircraft, automobiles, nuclear power plants are a few interesting examples. The benefits to society are many: cheaper, better, faster products that prove useful to humans.

If society had little desire for such progress, safety could be reasonably assured [1]. Long established, safe designs could be carefully improved in tiny increments and proven processes could be instantiated for development and production. Verification techniques could be highly refined. Technical progress would then be limited by the developers' ability to fully understand and predict the behavior of their creations. This limitation would slow progress to a near halt. The expense involved in designing provably safe products would limit their utility, practicality and availability to mass markets. The allocation of resources devoted to technical progress would be severely limited.

Society does not require perfect safety. We are willing to sacrifice a measure of safety for more performance, versatility and economy.[1] Many risk tradeoffs are acceptable, but not all are. The law of tort, in recognition of greater social goals of progress, does not always allocate the costs of accidents to the product developers. When the activity that led to the accident is considered more socially valuable than the risk, the developers may externalize the costs of the accident, the victim is left to fend for himself. However, if the social value of the danger outweighs its benefits, the developers may be forced to internalize the costs of the accidents. Socially irresponsible risk-taking and innovation are given an economic disincentive in the market [2].

Software is a relatively new artifact to enter the world of risk. It appears to be novel among other artifacts of the design activity: it doesn't break or wear out, it is not easily visualized in physical space, it consists of machine executable instructions and not physical components. It has been used to control nuclear power plants, commercial avionics, automobile safety systems and medical therapy devices. For instance, in the most widely publicized software product accident to date, the Therac-25 medical linear accelerator caused several deaths and injuries over a two year period. An FDA investigation revealed that the software played a significant role in the accidents [3]. Software designers know that they cannot guarantee the safety of these systems, even with unlimited resources [4]. As society moves toward increased use of computer control of safety systems, we should expect accidents and resulting lawsuits.

Two main levels of tort liability have been developed to deal with personal injuries due to the products of engineering: *strict liability* (liability without fault) and *negligence* (liability based on lack of due care). The different risk management implications of each legal standard are significant and are discussed elsewhere [5]. Suffice it to say that the difference is very important to our software engineering processes.

[1] Just consider the highway speed limit. If society wanted near perfect safety, the limit could be lowered to 10 m.p.h. This is unlikely to happen.

There is yet no common law decision that defines the level of liability for different classes of software defects. However, the law of products liability has established its own classification of product defects and through common law cases, has developed several critical characteristics to distinguish the standard of liability for a given defect. In this paper, we discover and assemble the relevant distinguishing characteristics that are used to classify product defects. We then propose some common types of defects in code where specifications are not correctly implemented. The legal characteristics are then applied to the code defects to determine the proper standard of liability. It is shown that, for this narrowly defined set of code defects (code "mistakes"), strict liability is the proper standard to be applied in cases where personal injury results.

2. NOTIONS OF DEFECT

We first note two distinct notions about "defect." The first notion is based on social requirements for worth or utility and is expressed in reasonable user expectations.[2] The second notion points to a more technical idea about whether the product, as implemented, satisfies its technical requirements and specifications. This notion is captured in a software engineering text as a "deviation of the observed behavior from the specified behavior" [7]. Notice that both notions of defect appear in software. The first is found in cases where unexpected behavior of the software is rooted in the specification itself, not a deviation from it.[3] The other form of defect, the concern of this paper, is based on a standard internal to our development processes: does the implementation satisfy its own specification? Did we build the product we planned to build? Did we turn out a random variation that injured someone?

3. THE PROBLEM

Our focus is code defects wholly unrelated to engineering judgment. We only consider "mistakes" made by the programmer in implementing a well defined specification.[4] Defects such as "off by one", incorrect variable initialization and passing an incorrect parameter to a function can be viewed as simple human mistakes.[5] We do not know how to completely avoid such defects nor can we test to find every one [8]. Such is the problem of any large, complex development project.

The thing about software that exacerbates this problem (common to many other kinds of products) is that programs are binary in nature, that is, a "small" mistake in the code can have unpredictably large consequences [9]. Thus, simple coding mistakes such as those discussed here can have serious consequences,. They've been shown to be contributing factors in particularly horrible personal injury cases [3]. They continue to plague the industry despite our best efforts.[6] We find that the law generally places embedded safety-critical software under the "products" rubric [10]. We also know that, in personal injury cases involving products, license disclaimers have no legal effect.[7] Software developers must respond and defend themselves from lawsuits where software appears to be a contributing cause to an accident involving personal injury.

How will this affect our safety-critical software development processes? What are cost effective measures that can be applied to help reduce the risks of large liability awards due to simple faulty coding? These questions motivate this paper. In particular, we ask the question, "*what legal standard is applied to failures of code to meet our software specifications for safety*?" It sounds like a legal question and it is. However, it is equally a technical question. The question is important since two very different legal standards may be applied depending on the class of the defect. One standard, *negligence*, allows the defense of "due care" and focuses on the adequacy of the development process. The other standard, *strict liability*, allows liability "without fault" where due care and the quality of the development process is irrelevant. The negligence standard will be raised when the "foreseeable risks of harm … could have been reduced or avoided by the adoption of a reasonable alternative design …" (a design standard). The strict liability standard will be raised when "the product departs

[2] This sort of defect appears in the design of automobiles whose design is not considered "safe enough" by social standards: an economy car that has a gas tank placed where rear-end impact is quite likely to cause explosion is considered defective. This is bolstered by the fact that an inexpensive modification to tank placement would increase safety by orders of magnitude [6].

[3] This can result from mismapping real user requirements into the specification or making particularly poor design decisions respecting the users in general. In that sense, it is a high level validation issue: validating software specifications to the requirements of society at large. Other research shows such social requirements can be derived from the law of tort. See generally [2].

[4] A specification that is complete, consistent and correct respecting its subject matter. Note that this paper does not deal with a programmer's intentional "redesign" that conflicts with a safety design specification. That is for another paper.

[5] Late nights, little sleep, too much coffee, stress: common to many programmer's lives. These things are known to contribute to mistakes in code.

[6] This is nearly universally recognized, see [4] for explanation. Also see the Risks Forum on USENET for current discussions and examples of such accidents.

[7] This is long established when dealing with personal injuries due to faulty products. See [11], [12], and [13] for further details.

from its intended design …” [13] (a product standard). These statements of law instructive, but are broad summaries of general principles. Therefore the authors searched the literature and common law cases for specific factors and analogies used in actual court cases to distinguish the classes of defects (and standards of liability.) These factors will be used to place simple coding mistakes into the proper classification and determine the appropriate standard of liability.

4. LITERATURE

Though not focused on the particular issues raised here, there has been some discussion of strict liability for defective software in the literature. One engineer makes the claim that unintentional failures to meet software specifications in code would rarely (or never) occur because of the precise syntax of programming languages [14]. This argument has some merit as far as the progress we’ve made in creating programming languages that obviate a few types of code defects. However, this progress cannot eliminate such defects in code as will be shown in the examples.

Some authors argue about whether such a strong legal standard, strict liability, should apply to software at all. They cite distinctions between software and other, more traditional products as a basis for the arguments. These arguments have not yet been answered in all Courts but the weight of the evidence is in favor of the application of strict liability where software is used to control a safety-critical embedded system. Some explain the applicability of strict liability to software by the fact that the injury may be caused by a product that was more dangerous than it was designed to be. This sort of defect yields no social or technical progress so there is no reason to allocate the costs to the victim [10].

The authors have further found that much of the discussion about software liability is clouded by syntactic problems with words such as “product” and “manufacturing” as used in the law. The actual semantic content of these legal terms is not constrained by simplistic associations with older technologies.[8] We make an effort to keep the vocabulary clear in this paper.[9]

[8] Terms such as “product” and “manufacturing defect” often carry connotations of older technologies while the common law of tort considered here depends on underlying principles, not mere names. Note that the situation is similar in the software engineering domain: the term “window,” for example, is not made of glass, does not shatter. The term used in software carries the older concept of glass window to a new context.

[9] In particular, we do not use the term “manufacturing defect” as applied to software code. The underlying concepts apply perfectly naturally, though.

5. LEGAL STANDARDS, SOFTWARE REALITY

Though the law of products liability and its defect classification has not yet been applied directly to software, there is a rich history in caselaw where defects are distinguished in other sorts of products and full explanation is detailed. Five basic factors found in the law have been listed below with a short explanation for each. The factors may not be disjoint, however, the authors found them useful at least as alternative ways to think about classification of product defects in software.

- <u>Standard Used for Comparison</u>: Design (exhibited in specifications) is judged “defective” by a social standard. The Court or jury must decide whether the engineers’ design intention reasonably balances social risks and utility. In contrast, mistakes in implementation are found by comparison of the product to the developers’ own technical standards [15]. If the product is defective by internal technical standards, it is more dangerous than it was designed to be! [12]

- <u>Degree of Human Intention</u>: Design defects may be avoided by a socially responsible risk-utility consideration during design. Mistakes in implementation cannot be eliminated this way, they are not the result of “consideration” of alternatives at all, but are failures in the process of construction of the product [12].

- <u>Avoidability of the Danger</u> Design defects in specifications involve conscious decisions of the design engineers. Mistakes in implementation are not the result of conscious decisions but of inadvertence. The developer knows that a certain amount of imperfection results from the construction process, regardless of the care taken in quality control [12].

- <u>Defect Visibility</u>: Design features define the product’s functionality. Thus, any defect in design is a consciously chosen characteristic. In this sense, the defects are “known” and “visible” to anyone who understands the product. Mistakes in implementation are not seen or known since they are latent. They are unplanned product “features.”

- <u>Consumer Participation in Risk Reduction</u>: Some design features include necessary risks in their beneficial use.[10] Consumers may then

[10] Consider the knife. Should manufacturers of knives be held liable when a consumer gets cut by the knife? The cutting is the feature the consumer desires from the

participate in risk reduction in order to enjoy the product's benefits. Mistakes in implementation are latent defects and consumers cannot generally participate in risk reduction [16].

The factors found above are summarized in table 1.

	Failure to meet social expectation	**Failure to meet own specification**
Standard used for comparison	external, a social standard for risk-utility decisions	internal, the developer's own standard is considered
Degree of human intention	conscious decision of the design engineers	inadvertent, "mistake"
Avoidability of the danger	avoidable by proper risk-utility consideration	unavoidable
Defect "visibility"	visible part of functionality, a planned characteristic of the product	latent, not known before the accident (or QC would have rejected!)
Consumer participation in risk reduction	sometimes consumer found "best" risk avoider	not possible because defect is latent
	NEGLIGENCE **Possible utility might justify allocation of damages to victim.** ***(Due care is central issue: focus on adequacy of process.)***	**STRICT** **No utility in building product "more dangerous than designed."** ***(Due care not relevant: focus on adequacy of product.)***

Table 1

product, and is expected to use it with care to minimize the chance of accident. This is a very simplified analysis but does make the point.

This table may now be used in the evaluation of specific coding defects to see what standard of liability should be applied to the coding defects of listed below. We list several mistakes in implementation of the specification that we've often seen in code examples.[11] The code given is meant for example only and follows the style of the C, C++ or Java. Such mistakes may be seen to be language style dependent: some languages provide opportunities for certain sorts of mistakes while preventing others. We believe that C style languages are prevalent enough in safety-critical software development that we should consider them. Therefore, we presently restrict our attention to these examples.

Consider the examples for the general errors in logic that they present if they are not immediately discovered and corrected:

1. x = x + 5 instead of x = x * 5 (arithmetic)
2. x <= 5 instead of x < 5 (logic)
3. wrong variable name: x = b * 5 instead of x = c * 5 (arithmetic)
4. x = x + 1 instead of x = 1 (one of the Therac code defects, may lead to "buffer overflow.")
5. loop counter variable reset in the incorrect place.
   example: while (i < 5) { code.... i = 0; }
   instead of:
   i = 0; while (i < 5) { code ... }
   (serious problems, easy mistake!)
6. array off by one: declaring an array of length 11 instead of 10 - using a language that starts counting at 0 for index values. (simple human assumptions)
7. crossing of arguments when calling a function.
   given a function prototype: int func (int i, int j) the developer may give an integer for i when it was meant for j and vice versa. (unpredictable results)

These mistakes look like they should only be found in first-year programming class projects, yet it is their simplicity that makes them so easy to overlook (and hard to find) in large projects. The characteristics of the defect classification can now be applied to these mistakes to determine within which class they fall.

Mistake number 1 will be the primary example from which all other mistakes will follow. The mistake is found only by comparison to the developer's own standard. There really is no social standard for what I can do with arithmetic. It falls in second column of row 1. When analyzing the "degree of human intention" in the second row of the table, it is clear that these defects are not intentional, they are "mistakes." Again, it also falls in the second column of row 2. Can the designers of the

[11] We believe that these particular kinds of code mistakes are not always found by testing or inspection, though some portion of them are likely to be found.

software now alleviate the danger by some sort of safeguards? Not reliably, since we don't know the nature of residual mistakes like this. Once again, it falls in the second column of row 3. Is this defect the sort of thing we can observe and analyze, is it "visible?" If it was, quality control would have it corrected! It is a latent defect and again, falls in the second column of row 4. Finally, can the user (the consumer) guard against the danger? The user can't read about it in the manual and has no way to know what precautions to take. It falls in the second column of row 5. Defect number 1 contains each and every identifiable characteristic of a legally defined failure to meet product specifications. This results in the application of the strict liability standard. The product is considered "more dangerous than it was designed to be!" Note that examples 2 – 7 produce precisely the same results.[12] If any defects like these remain in our code and are found to be a substantial cause of personal injuries to a user, we are liable. Our "due care" is irrelevant.

6. WHAT CAN WE DO?

First note that there *are* defenses to a claim of strict liability. The particular defect must be proven to be a "substantial" or "proximate" cause of the damages. The product may have been legally "misused" or modified by the user so that the cause of the damages is not really the developer. These issues are important, beyond the scope of this paper, and well discussed elsewhere [11].

Mere process, by itself, cannot solve this problem. Liability is assessed when proof is found showing the defect that caused the accident was a failure to meet the developers own specification. ISO 9000 or CMM certification, relevant to a negligence case, has no relevance here. The only hope to avoid this standard of liability is to avoid the defects themselves. Full verification is required to realize this hope. While we can always commit more resources to this effort and improve, full verification is not possible for nontrivial software systems [8]. The goal then, is to lower actual risk if and when possible. Then, lower the risk of catastrophic damages *when* undiscovered defects result in software failure. Thus, process is important to the extent that it:

- Actually lowers risk of mistakes in implementing safety specifications; and,
- Produces systems tolerant of mistakes in implementation of safety specifications.

[12] This is not to say that these defects could never be the result of intentional design decisions on the part of the programmer. There are some cases where arguments may be adduced to show the specification *is satisfied* by such code even though the "correct" version would have been safer. We do not cover these cases in this paper.

The first item calls for developers of safety-critical software to carefully gather and assess data on the efficacy of their own processes for reducing the number of mistakes in the implementation of their safety specifications. Processes that incorporate testing at every level of development aid in reducing the number of actual defects. Mistakes can possibly be found during unit testing and even during black box or user testing. No matter how much testing is done, nothing can guarantee that no defects in code remain [17]. It is best that our business models include a risk management component [5]. In this way, contingencies may be planned that will avoid the possibility of a simple mistake bankrupting the development organization due to liability costs.

The second item is another important topic with its own literature, a good overview is given by Leveson in [4].

7 S.HORTCOMINGS AND FUTURE WORK

This work is part of a larger project we have undertaken at Cal Poly State University. Our overall goal is to determine the particular requirements of the law of products liability and design processes that address them adequately. We have the further goal to at least partially automate these processes by use of workflow technologies. We have already done work investigating the negligence standard for design defects and generated a basic model for workflow assistance in lowering risks of liability [18]

In writing this paper we found it particularly difficulty to account for all the legal arguments and technical arguments that might be relevant. We did our best to cover the major arguments and to reference sources to fuller discussions, well beyond the scope of this paper (or even a book!)

Our mistake analysis is done in a C like language. We have not considered other (different) languages such as Lisp, where things would be different. We plan to extend our analysis to languages relevant to safety-critical software and will assess more of them as needed.

8. ACKNOWLEDGEMENT

We gratefully acknowledge the encouragement and generous support of Mr. Jon Simonds and his wife Olivia in this work.

REFERENCES

[1] Petroski, To Engineer is Human, Vintage Press, NY, 1992

[2] Turner, Richardson, Software and Strict Products Liability: Technical Challenges to Legal Notions of

Responsibility, *Proceedings of the IASTED International Conference on Law and Technology*, San Francisco, Oct. 2000.

[3] Leveson, Turner, An Investigation of the Therac-25 Accidents, *IEEE Computer*, *Vol. 26* No. 7, July 1993.

[4] Leveson, Safeware, Addison-Wesley, 1995.

[5] Turner, "Risk Management for Safety-Critical software: A Unique Problem on the Horizon," published in *The Technology Report*, a publication of the Technology Section of the Adademy of Legal Studies in Business, 2001. The paper is currently held at: www.csc.calpoly.edu/~csturner/research.html

[6] Grimshaw V. Frod Motor Co., 174 Cal. Rptr 348 (Cal. App. 1981).

[7] Bruegge, Dutoit, "Object-Oriented Software Engineering: Conquering Complex and Changing Systems." Prentice Hall, p 517 (2000)

[8] Kaner, "The Impossibility of Complete Testing," *Software QA*, vol. 4, no. 4, p 28 (1997).

[9] Hamlet, "Are we Testing for True Reliability?" *IEEE Software*, July 1992.

[10] Turner, "Software as Product: The Technical Challenges to Social Notions of Responsibility," Ph.D. dissertation, Department of Information and Computer Science, University of California, Irvine, 1999. Available at www.csc.calpoly.edu/~csturner/research.html

[11] Witherell, How to Avoid Products Liability Lawsuits and Damages, Noyes Publications, 1985

[12] Prosser, Keeton, *Prosser and Keeton on Torts, 5th Edition* (West Publ, St. Paul, MN. 1984).

[13] Restatement Third, Torts: Products Liability, American Law Institute Publishers, MN, 1998

[14] Hecht, "Products Liability Issues for Firmware in Consumer Systems," submitted to Professor F. Olsen, UCLA School of Law and to Professor C.S. Turner of Cal Poly State University, unpublished manuscript, 2001.

[15] Prentis v. Yale Mfg. Co., 421 Mich. 670, 365 N.W.2d 179 (Sup. Ct. MI, 1984).

[16] Henderson & Twerski, *Closing the American Products Liability Fontier: The Rejection of Liability Without Defect*, 66 NYU L. Rev. 1263 (1991).

[17] Parnas, Clements, "A Rational Design Process, How and Why to Fake it," *IEEE TSE*, Vol SE-12, P 251 (1986)

[18] Turner, Khosmood, "Rethinking Software Process: the Key to Negligence Liability," Proceedings of the Fifth IASTED International Conference, Software Engineering and Applications, August, 2001, Anaheim, CA.