UNIVERSITY OF CALIFORNIA

Irvine

Software as Product: the Technical Challenges to Social Notions of Responsibility

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy in
Information and Computer Science

by

Clark Savage Turner

Committee in charge:

Professor Debra J. Richardson, Chair

Professor John L. King

Professor David F. Redmiles

Signature Page goes here

Dedication

To

Yusuf Marcus Turner

and

Hannah Rose Turner

in recognition of their worth as human beings

and as wonderful children to father

an apology

If the fool would persist in his folly he would become wise.

William Blake
“Proverbs of Hell”

Contents

List of Figures

Acknowledgments

I am fortunate to have Professor Debra Richardson for my committee chair. Without her unwavering support and encouragement I would not have continued my work and finished. Without her keen insights and criticisms, I would not have finished so well.

I am deeply indebted to Professor John King for his time and patience (of several years) in helping me to narrow my research efforts down to a reasonable scope. His humor and his passion for his work often inspired me to continue working with my own ideas. Professor David Redmiles provided an attentive ear and valuable feedback when I most needed it.

A host of others provided comments and suggestions central to the outcome of this work. David Wright McDonald discussed my work with me on a weekly basis and provided warm friendship and useful insights. Arthur Reyes gave of his time and his heart with his comments and encouragement. Roger Neyman of CMD Technologies provided his insights to ground my academic vision in industrial experience. Cem Kaner, that tireless advocate for software quality, provided early support and access to his excellent library.

In addition, I am grateful to Lavonne C. Pineda, a Chiropractic Intern who helped support and care for my committee chair through the challenges of working with me.

Perhaps the biggest thanks of all go to Dr. Belinda Morrill. She has been my wife and sparring partner through this long journey. Without her help, I might never have realized that my only writing blocks were inside me.

Finally, thanks to my men's group: Vivi, Doug, Brian, and Roger. Fare well.

Curriculum Vitae

Clark Savage Turner

1979	B.S. in Mathematics, Kings College, Wilkes-Barre, Pennsylvania
1981	M.A. in Mathematics, Pennsylvania State University, University Park, Pennsylvania
1986	J.D., University of Maine School of Law, Portland, Maine
1993	M.S., Information and Computer Science, University of California, Irvine
1999	Ph.D., Information and Computer Science, University of California, Irvine Dissertation: "Software as Product: The Technical Challenges to Social Notions of Responsibility" Professor Debra J. Richardson, Irvine, Chair

PUBLICATIONS

Turner, Richardson, King, "Legal Sufficiency of Testing Processes," *Proceedings of the 15th International Conference on Computer Safety, Reliability, and Security*, Vienna, Austria, October, 1996

Leveson, Turner, "An Investigation of the Therac-25 Accidents," *IEEE Computer*, Volume 26, Number 7, July, 1993

Kling, Turner, "The Information Labor Force," published in Kling, Olin, Poster, *Postsuburban California: The Transformation of Orange County Since World War II*, University of California Press, 1991

Sganga, Turner, "Orange County Lawyer's Salary Survey," *Orange County Lawyer*, September, 1991

Abstract of the Dissertation

Software as Product: the Technical Challenges to Social Notions of Responsibility

by

Clark Savage Turner

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1999

Professor Debra J. Richardson, Chair

Products liability issues are virtually certain to arise for software products in the near future. Their resolution can have enormous impacts for the software industry. In particular, code defects must be classified into those of design intention and those of inadvertence in construction. The distinction between these two classes of defect is crucial to the operationalization of current social notions of responsibility for accidents through the law of products liability in tort. This work finds that such distinctions depend on software specifications alone, unlike for other manufactured products. Further, such distinctions cannot be made for arbitrary code defects. A model of software production is developed to explain the unique nature of software that presents this new challenge to the law. The model further demonstrates that

the use of specifications to distinguish the class of a code defect is a subjective exercise. Thus, the distinctions are of little use in assignment of responsibility for software related accidents.

Chapter One:

Introduction

“Cheaper, faster, safer. Choose any two!”

Homo Faber, “man, the maker.” People build things, new things that have not existed before. Things that change the world. Designers use science, experience, and intuition to create new artifacts with desired properties. Aircraft, automobiles, nuclear power plants are but a few examples. The benefits to society are manifold: cheaper, more efficient transportation, new sources of energy.

If society had little desire for such technical progress, safety might be reasonably assured. [Pet92] Long established, safe designs could be carefully improved in tiny increments. Verification techniques could be highly refined. Technical progress would then be limited by the designers' ability to fully understand and predict the behavior of their artifacts. This limitation would slow such progress to a snail's pace. Further, the expense involved in creating fully safe designs would limit their usefulness, practicality and availability to large markets. This would bound the allocation of resources devoted to technical progress. For instance, would air travel have developed to its current state of technology (and economy) if accidents were not to be tolerated?

Alas, society is unlikely to tolerate the arrest of technical progress in exchange for complete safety. We want more, better, faster and cheaper. We are sometimes quite willing to sacrifice a measure of safety for more performance, for more versatility, or for economy. Many risk tradeoffs are socially acceptable, but not all are. [Pin81]

Tort Law

The presence of risk in design artifacts has serious social consequences. A tire blowout at high speed on a crowded freeway can result in human disaster. Such disasters cause economic damages and hardship to the victims. Older notions of justice would hold the artifact's creators responsible as the "cause" of the victim's damages. [Wit85]

However, such a notion strictly enforced on an industry would prove too expensive and limit efforts at technical progress. This would correspondingly limit the social progress in transportation, medicine, energy and other fields that are enabled by technical progress.

The law of tort, in recognition of greater social goals of progress, does not always allocate the costs of accidents to the creators of the artifact. The goal of this area of law is to maintain a reasonable social balance of risk and benefit by its allocation of costs due to accidents. [Pro84] When the activity that led to the accident is socially valued more than the risk, the creators of the artifact may externalize the cost. The

victim's interests are sacrificed for the good of the greater society, and he bears the costs alone.[1] However, if the social value of the risk outweighs the benefits, the creators of the artifact may be forced to internalize the costs of the accident by compensation to the victim. This scheme allows for a reasoned economic advantage to be given to designers of artifacts that promote long term social welfare. It is illustrated by simple diagram in Figure 1 below:[2]

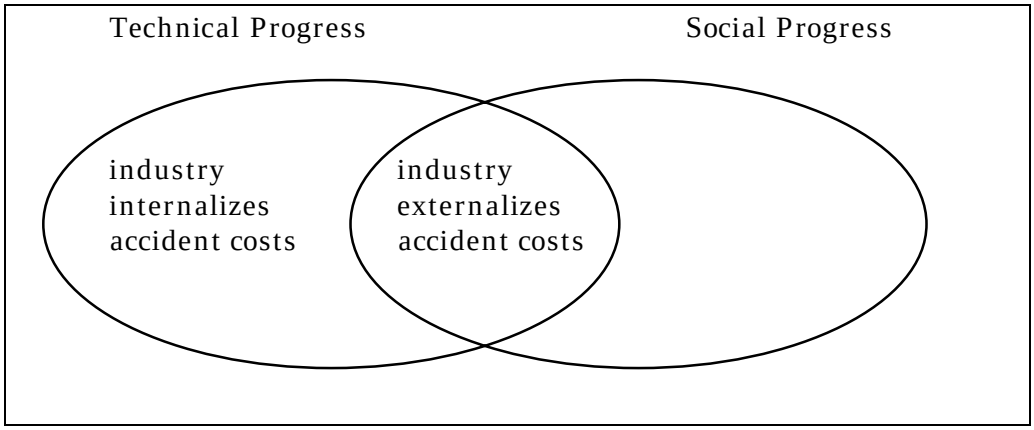

Figure 1 - Social Risk Management

Products liability in tort deals with artifacts of design as they have been discussed here. It implements the social risk-benefit analysis by its declaration of "defective" artifacts, called "products," those undeserving of social support. The legal notion of defectiveness is therefore an operationalization of the underlying social goals of long term welfare.

[1] Church, family and other social welfare programs may ultimately spread these costs through society.
[2] This diagram is generally attributable to [NW82].

Consider the following two basic notions of product defectiveness.

1. Defects of *design intention*: the normative judgments of the designer are under scrutiny. Was the risk considered adequately, are the benefits worthwhile? A declaration of defect is a legal conclusion based on the risk-benefit analysis.
2. Defects of *manufacture*: the ability to flawlessly reify design intention is at issue here. Does the product itself embody the design intention or is it some unplanned variation? This is not a normative issue, it is a descriptive one. A defect in manufacture is declared based on technical, descriptive standards.

Enter Software

Software is a new artifact of design to enter the world of risk. It appears to be novel among other design artifacts: it doesn't break or wear out, it is not easily visualized in physical space, it consists of machine instructions and not physical components. It has been used to control nuclear power plant safety systems, commercial avionics, automobile brakes and medical devices. Flaws in such software can cause or contribute to personal injuries. 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. [LT93] Software designers know that they cannot guarantee the safety of these systems, even with unlimited resources. [Lev95] As society moves toward increased use of computer control of

safety systems, we should expect more software accidents and resulting lawsuits. [Web92]

There is yet no common law that classifies defectiveness for software products, so when lawsuits were filed in the Therac case, the parties could not reasonably predict the outcome. Before the judge was forced to face novel legal questions about defectiveness of software products, the suits were settled to the satisfaction of the several victims or their survivors. No legal precedent was set.

The answers to the novel legal questions will take the form of definitions of product defects in the software domain. They will have major economic implications for the safety-critical software industry. If "bad" law is declared that does not accurately reflect social goals or applies them in an irrational manner, the costs to society and the industry can be enormous. Though common law is self-correcting, change may be quite slow relative to the rate of change for software technology.

The Research Question

The research hypothesis for this work is that the extant legal notions of defectiveness may be rationally applied to the software product by analogy to a stage of production model. The analogy views coding as a part of software product "manufacture" and necessitates the reliable identification of unintentional defects in "construction" of code.

The research question is thus, "is it possible to reliably identify defects in manufacture for software code?" The answer is, unfortunately, "no." Software is a unique product in that mistakes in construction cannot always be distinguished from design intention by any currently known legal or software engineering method. This looks like a legal question. It is. But the legal and social understandings that lead to rational normative rules depend completely on accurate descriptions of the software artifact. Ultimately, it is the software designers who will explain the nature of their artifact to society and assist in the process of creating accurate understandings and working rules to implement social goals.

Contributions to Scholarship

This dissertation makes several contributions to scholarship in software engineering. It helps to demystify the legal obligations placed on software products by products liability law. The notion of requirements of the "remote customer" places legal obligations into the context of software product requirements in a natural way. Such obligations considered as requirements enable explicit consideration at an early stage in the development process, when the cost of defect discovery and removal is possibly at its lowest. Software risk analysis benefits from this work insofar as the analysis reveals the elements of expected cost for different classes of defects. Further, a simple model of software development is described that gives insight into the known difficulty of separating software design from implementation.

Legal scholarship and the common law will also benefit from this work. Activity in the common law court's development of accurate and fair rules can take many years of individual cases bringing a single issue at a time (or a very few issues) for the court's consideration. Part of the value of this dissertation is to bring much of the specific contextual information to the attention of courts, researchers, expert witnesses and attorneys before the activity begins, which can help guide the common law to a speedier and more accurate resolution of the social obligations into rules for software.

This dissertation is of special value in that courts must be reactive to specific problems that arise, but this work is initiated proactively. Similarly, this work is very general and objective in nature, it is not restricted to the particular arguments brought by parties with specific motivations in a court case. It can be of much broader value than individual case precedent.

Overview of the Dissertation

The chapters and the basic ideas of each chapter are outlined below:

Chapter 1: Introduction. Introduce the classification of product defects. Ask the generalized research question and provide an answer.

Chapter 2: Related Work. Legal and Software opinions surrounding the question are explored.

Chapter 3: Products Liability in Tort. Introduce the law of products liability. Give formal details of the defect classification. Explain the essential distinctions between the classes of defect. Criticality of the distinctions is explained.

Chapter 4: Constructing Software Code. A general description of the job of the coder is given relative to possible software flaws. Process models are used to illustrate the software notion of design as distinguished from coding.

Chapter 5: Defects in Software Code. Apply the law explained in chapter 3 to the reality of the code as explained in chapter 4. Find that the defect classifications are sometimes impossible to apply because of the nature of the software product.

Chapter 6: Analysis. Compare software production to the production of automobiles and search for the inherent differences that might explain the failure of the defect classifications. Present a model that explains it. Show that software engineers' separation of design and code is a subjective exercise.

Chapter 7: Conclusions, Consequences and Future Work. Detail future directions and questions generated in the process of this work

Chapter Two:

Related Work

This investigation of defects in safety-critical software products necessarily includes related work from two main bodies of knowledge:

- Software Engineering; and,
- the Common Law of Products Liability in Tort.

The reader should notice that these two bodies of knowledge have their own working vocabularies, and the intersection of those vocabularies is not empty. In the coming chapters, specific definitions will be detailed. At this stage, general notions are sufficient.

Legal Research

Law review articles going back as far as 1979 show legal concern about the injury potential for defective software products. [Nyc79] Several scenarios of software related injuries are posed and legal liability discussed in subsequent articles published in the 1980's. [Gem81] [BD81] These writers exhibited a fair understanding of the software design artifact as understood at the time. Later

articles by other authors continue to rely on those early analyses and have become somewhat dated in the maturity of the software analyses. [Mor89]

Legal Notions of Software Defects in Manufacture as Distinguished from Design

Legal commentators exhibit a full range of opinions and reasoning about the existence and definition of a rational distinction between defects in software design and manufacturing. Wolpert implies that software *is* design and that manufacture "[i]s simply a question of making copies onto appropriate media." [Wol93 at 524] Mortimer explains that, "[a] manufacturing defect occurs when products are not produced as designed. After a system is designed, it must be encoded into a workable program. If an error was made during data input or conversion from source code, or if a flaw was discovered in the physical diskette or hardware, a manufacturing defect would occur." [Mor89 at 190] Brannigan and Dayhoff define manufacturing defects for software so that "ordinary mistakes by programmers in carrying out the system designer's instructions would qualify..." They further wonder whether a rational distinction between the defect classes exists because, "it may be difficult to distinguish the design phase from the production phase." [BD81 at 138] Cronin writes that "[d]esign errors....comprise the most serious class of defects in mass-marketed computer software," but recognizes that a sharp definition of software "design and assembly" may be impossible, "[p]rogammer discretion within the confines of a program's design resembles both design and assembly..." [Cro85 at note 124]

Miyaki [Miy92] is more concerned with the overarching social goals that must be supported by any distinction. He says that although California Courts might justifiably impose some particular definition to apply products liability to software, they should not do so because of the adverse impact it would have on software innovation.[3] Thus, the distinction should not be made at all for software because of larger policy reasons.

Software Engineering

Software Engineering Concerns for Social Expectations for Software Product Defects

Software research has recently shown serious concern for the safety of software controlled physical systems. [Lev95] [Par90] In general, the social and legal obligations on software development are given only passing mention and are viewed as entirely secondary concepts to the technical issues. For a typical example, see [Lev95] and note the absence of an entry for "liability" (or other related term) in the index. Some liability issues receive passing mention but indicate a rather naive understanding of the field of tort law, such as [Voa97]. Voas titles his paper, "A Crystal Ball for Software Liability," but presents precious little information about the actual legal requirements involved in testing to avoid liability. He claims that software liability, "stems directly from three classes of problems: erroneous input data (from sensors, humans, or stored files), faulty code, or a combination of the

[3] This comment belies his mistaken belief that products liability is ultimately strict liability,

two." [Voa97 at 31] He is, of course, referring to failure as the sole and direct cause of liability. This is not true in all cases, so the claim is misleading. Nowhere does he introduce or discuss the legal distinctions between defect classes and the resultant expected cost disparities.[4]

Software Engineering Notions of Software "Manufacture"

First, note that the discussions of "manufacturing" within the software engineering field commonly depend on engineering understandings of the term "manufacturing" and are not related to the precise legal definitions to be developed in the next chapter. Further, the term "manufacture" usually arises in the context of making distinctions between software production and the production of other design artifacts. I include the ideas discussed below to illustrate the software engineer's basic understandings of software defectiveness.

Only a few software researchers have spoken directly to the identification of defects in the "manufacturing" of software. Parnas sees defects (errors) in software as not statistically independent, which distinguishes them from random defects introduced in the manufacturing of other traditional products. He goes on to identify the compiling phase of software development as manufacturing, "in software, there are few errors introduced in the manufacturing (compiling) phase." [Par90 at 638] Hamlet [Ham92] refers to software defects as mostly defects of design, where defects in the *physical machine or storage media* remain to be considered as defects

---

a premise that is not correct. This is discussed in the next chapter.

[4] This is not a criticism of his basic technical contribution to risk quantification techniques!

in manufacture. Leveson claims that for software, the "manufacturing phase is eliminated from the lifecycle," but comments that "duplication of software might be considered to be manufacturing." [Lev95 at 22] However, note that some software research refers to code implementation as "construction" of the software product. [GG75] [Som92] This notion could be analogized to product manufacture, allowing for defects in manufacture during the coding activity.

Summary of Important Points

Both fields show some concern for the potential for software artifacts that can threaten public safety. Legal research uniformly foresees products liability lawsuits but does not agree on whether both classes of defect are applicable to software code. Two authors of the early 1980's [Gem81] [BD81] anticipate the research question for this dissertation but do not analyze the possibilities. The software field, working from common engineering notions of manufacturing and design, generally refers to code as design, but also refers to the coding activity as "construction."

In the following chapters, these possibilities will be tested against precise legal definitions and known characteristics of a manufacturing defect.

Chapter Three:

The Law of Products Liability in Tort

"Safety is much too important to be left to the designers" [TWDP76]

Several terms used in this chapter have very specific legal meanings but are also commonly used in the field of software engineering. Specific terms that may cause confusion are:

- product
- design
- specification
- manufacture
- defect

The terms will be defined individually as needed. They are used in this chapter within the context of the legal framework in which problematic software will eventually be judged.

The Common Law

Products liability in tort is a part of the common law. Common law is judge-made law, to be distinguished from legislated, statutory law. Its hallmark is an ability to evolve with changing social conditions to meet current social needs, "a continually more efficacious social engineering." [Pou49] In this sense, it is self-correcting, capable of modifying itself to adapt to dynamic circumstances. A model of the common law process based on a standard process control model is given in figure 2 to illustrate the concepts.[5]

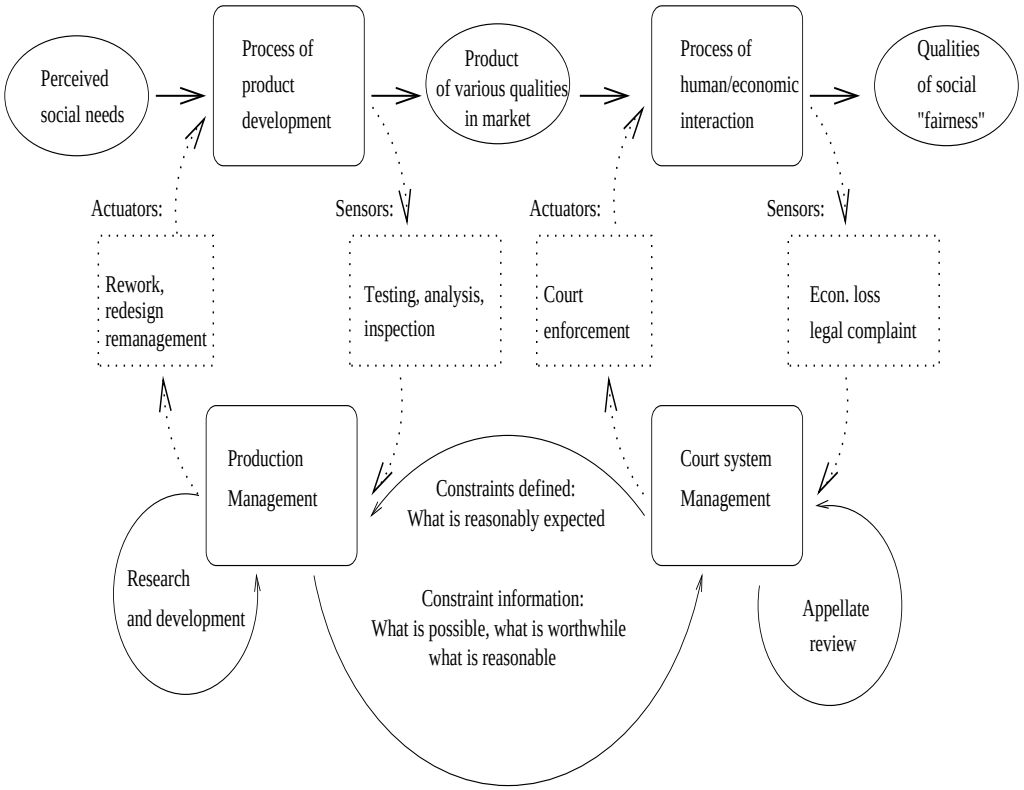

Figure 2 - Common Law Model

[5] This model was developed to explain the common law process in [TRK96].

The common law evolves on a case by case basis. When a products liability case involving a new issue comes up, specific rules to handle the issue are developed by application of fundamental tort principles to the concrete case at hand. The new rules that are developed this way must satisfy the following criteria:

- incentive for increased safety at a reasonable social cost; [Hen76]
- explainable, rational, repeatable method for decisions; [Owe80] [Eis88] and,
- terminal decision process. [Eis88]

Of course, satisfaction of these criteria depend on the reality of the domain of application as well as the precise formulation of the legal rule itself. For example, "incentive for increased safety" and "reasonable social cost" are terms that will derive meaning only in the domain where they are applied. The social expectations underlying the legal rules come from the legal system, but the technical possibilities and realities are properties of the domain of application.

Products Liability

The modern law of products liability developed from roots in negligence and warranty causes of action. In the 1960's *strict products liability in tort* developed in response to the perceived inadequacies of negligence and contract-warranty causes of action when applied to products of modern complexity involved in personal

injury.[6] It was to be based on proof of *product defect* rather than proof of *fault*. Once a product was proven defective, damages could be awarded. On the other hand, for a negligence case, fault (unreasonable conduct) must be proved, the injury may be merely economic in nature, and a product need not be the instrumentality of the injury (it could be a service).

Further, products liability cases in tort *are not subject to contract or license disclaimers of liability*. [Hen60] The commonly seen "limitations of liability" and conspicuous statements negating the manufacturer's responsibility for the behavior of the product have no affect in a products liability case for personal injuries.

Notice also that tort law should not be confused with criminal law. Tort law is civil in nature and does not normally result in incarceration.[7] Civil cases are merely a question of monetary damages. In fact, as long as one can afford to keep paying damages, one may continue with tortious conduct.[8]

There are two basic prerequisites for any case of products liability in tort:

1. personal injuries or harm to other property must be involved, "pure" economic damages will not support a case; [Pro84]

[6] For a good historical view of the development of strict products liability, see generally [Bir80]
[7] Civil contempt citations may result in fines and punishment, but this is really another area of law.
[8] Up to the point that it becomes intentional or has the quality or reckless disregard for the safety of others.

2. the instrumentality causing the injury must be considered a “product” or a product “component;” [Pro84] provision of services will not support a case.

The first of these prerequisites is self explanatory and depends on the facts of each case. The second involves a legal judgment that the design artifact will be considered subject to the law of products liability.

Products liability law is only applicable to a design artifact legally considered to be a product or is shown to be a product component. As with the term “manufacture,” the term “product” has specific legal meanings and is not restricted to dictionary definitions or common understandings of the term. [Flu85] Most legal discussions contrast “products” (to which products liability is applicable) to “services” (to which products liability is not applicable). Each of the following factors may be used to distinguish a “product” from a “service” for purposes of products liability in tort:

- whether the artifact has a value of its own as a tangible item;
- whether the artifact may be subject to “ownership;”
- whether defects can be corrected; and,
- whether the artifact may be mass produced and marketed.

[Mor89]

Since the law of products liability in tort is a branch of the common law, it may evolve differently in each of the 50 states. However, the overarching principles of

tort law by which states' rules are derived remain basically the same in all states. The American Law Institute has, for some years, prepared and published a standard text covering the basic tort principles and rules therefrom in its *Restatement of the Law* series. The Restatement is often considered authoritative for the common law and is cited by common law judges in tort cases. [Smi94]

In 1998, after several years of scholarly work and revision, the Restatement of the Law, Third Edition, Torts, Products Liability was published. [Res98] Section One of this text provides,

> **Section 1. Liability of a Commercial Seller or Distributor for Harm Caused by Defective Products**
>
> **One engaged in the business of selling or otherwise distributing products who sells or distributes a defective product is subject to liability for harm to persons or property caused by the defect.**

The central legal idea that triggers liability for products liability law is therefore product defect.

Product "Defect"

The Webster's Ninth New Collegiate Dictionary defines *defect:*

> ***1.*** **a: an imperfection that impairs worth or utility: SHORTCOMING [...]**
>
> ***2.*** **a lack of something necessary for completeness, adequacy, or perfection: DEFICIENCY [...]**

In general, then, a defective product will be one that contains some imperfection that makes it legally inadequate. This general idea has evolved through common law decisions for some time.[9]

As early as the 1200's, some forms of liability for *manufacturing defects* was imposed by the law. In the 1960's, American courts began to recognize that a commercial seller of any product having a manufacturing defect should be *strictly liable* in tort for harm caused by the defect. The liability is to attach even if the manufacturer's quality control in producing the defective product was reasonable! "Due care" of the manufacturer becomes irrelevant. Liability is based completely upon the product's failure to satisfy the manufacturer's own design intention. The product is "more dangerous than it was designed to be." [Pro84] Once the product is shown to have caused injury, the proof of a manufacturing defect is sufficient to result in liability.

In the late 1960's and early 1970's, questions of *design defects* began to arise when the product in question satisfied the intended design (i.e., not a manufacturing defect) but the design itself was unacceptably risky. In such cases, defects cannot be judged by reference to the manufacturer's own design standards because those are the very standards under scrutiny. The design defect involves a social judgment about the trade-offs necessary to determine which accident costs are more fairly and efficiently borne by those who incur them (the victims) and which are best

borne by product users and consumers through internalization of the accident costs (by the manufacturers) and having product prices reflect the relevant costs. [Pro84] Judgment of design defectiveness is often based on the availability of a cost-effective alternative design that would have prevented the harm. [Ban94] This is *not* a strict liability standard, but one more in the nature of *negligence*, based on lack of due care during the design process. [Bir80] It is a legal conclusion based on social standards for design adequacy. Injuries may indeed be caused by a design decision, but unlike the case of a manufacturing defect, if due care was exercised, the manufacturer is not held liable.

The *Restatement* distills the above into the following description of the law:

> **Section 2. Categories of Product Defect**
>
> **A product is defective when, at the time of sale or distribution, it contains a manufacturing defect, is defective in design, or is defective because of inadequate instructions or warnings. A product:**
>
> **(a) contains a manufacturing defect when the product departs from its intended design even though all possible care was exercised in the preparation and marketing of the product;**
>
> **(b) is defective in design when the foreseeable risks of harm posed by the product could have been reduced or avoided by the adoption of a reasonable alternative design by the seller or other distributor, or a predecessor in the commercial chain of distribution, and the omission of the alternative design renders the product not reasonably safe;**

[9] For a brief historical perspective of tort and the law of products liability, see generally [Res98]

> **(c) is defective because of inadequate instructions or warnings when the foreseeable risks of harm posed by the product could have been reduced or avoided by the provision of reasonable instructions or warnings by the seller or other distributor, or a predecessor in the commercial chain of distribution, and the omission of the instructions or warnings renders the product nor reasonably safe.**

This dissertation is limited in scope to analysis of the first two kinds of defects as they relate to software.[10] In general, defective warnings (the third category) are part of the product design and this is reflected in the similarity of the legal standard.

Essential Characteristics of the Classes of Product Defect

Commonly understood meanings of the terms "design" and "manufacture" may not prove sufficient to distinguish the categories of defect for legal purposes. The essential characteristics of the two categories have been defined and developed over time by common law decisions. Since the distinction between these two classes of defect is central to this dissertation, it is necessary to delineate the distinctions further. Five dimensions are used to illustrate:

1. Standard Used for Comparison: Design is judged "defective" by a social standard. The Court or jury must decide whether the design intention reasonably balances social risks and utility. In contrast, a manufacturing defect

is found by comparison of the product to the manufacturer's own technical standards. [Pre84] If the product is defective by the manufacturer's own standards, it is more dangerous than it was designed to be. [Pro84]

2. Degree of Human Intention: Design defects involve conscious decisions of the design engineers. Manufacturing defects are not the result of conscious decisions but of inadvertence. The manufacturer knows that a certain amount of imperfection results from the construction process, regardless of the intention to eliminate them by quality control. [Pre84]

3. Avoidability of the Danger: Design defects may be avoided by a socially responsible risk-utility consideration during design. Manufacturing defects 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. [Rix86]

4. Defect Visibility: Design features define the product's functionality. Thus, any defect in design is a consciously chosen characteristic for the product. In this sense, the defects are "known" and "visible" to anyone who understands the product. [Hen73] Manufacturing defects are not seen or known else they would have been removed during quality control. They are unplanned "features" of the product.

---

[10] The duty to warn has been questioned about its applicability to software. Warnings must be specific, and if a software designer knew of the specific risks, wouldn't the software be redesigned to reduce or eliminate the risk (as it relates to software)? See [GL81]

5. Consumer Participation in Risk Reduction: Some design features include necessary risks in their beneficial use.[11] In such cases, consumers must participate in risk reduction in order to enjoy the product's benefits. Manufacturing defects are latent and consumers cannot generally participate in risk reduction. [HT91]

The characteristics used to distinguish these defect categories are summarized below in figure 3. The defect class is given at the top of the table and the conceptual dimensions that have been used to distinguish them are given in the rows.

[11] 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 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.

	Design	Manufacture
Standard used for comparison	external, a social standard for risk-utility decisions	internal, the manufacturer's own standard is considered
Degree of human intention	conscious decision of the design engineers	inadvertent, a "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

Figure 3 - Defect Distinctions

Importance of the Classification

Besides an interesting academic exercise, what is the importance of understanding the distinctions in defect classes?

Perhaps most importantly, these distinctions operationalize social notions of responsibility surrounding design artifacts that cause personal injury. The development of innovative designs offer the possibility of technical advances that support social progress. So that society can make decisions about when to support design innovation and when not to, a negligence or "due care" standard is employed.

However, when an artifact does not embody design intention, the possibility of social progress vanishes and the costs of accidents are to be internalized to that activity. [Owe96]

The distinctions are also important to anyone potentially involved in such a case because, as explained above, the different defect classes are subject to a different legal standard of proof. For a manufacturing defect, the plaintiff-victim has a relatively low expected cost for prosecution of the case: it is simply a matter of proof that the product did not meet the manufacturer's own design standards. The defendant-manufacturer has a relatively greater expected cost for such cases since damages are awarded simply on proof that the defect caused the harm. Due care is no defense to such a case.

For a design defect, the plaintiff-victim has a much higher expected cost of prosecuting the case, as it must be proved that the defendant-manufacturer did not act reasonably given the state-of-the-art. This involves a much greater burden of proof, proving what the state of the art provides is often a matter of disagreement even between experts. Correspondingly, the defendant-manufacturer now has a chance to prove that its design was sufficient even though the injury occurred ("due care"), thus the expected cost of such cases can be lower than that for manufacturing cases. [Pro84]

Notice that, to the extent that manufacturing cases are seen as genuinely advantageous to victims, there will be pressure from the plaintiff's bar to

characterize defects as those of manufacture whenever possible. A high level flow chart of a products liability case is shown in figure 4 to illustrate the pertinent concepts discussed so far:

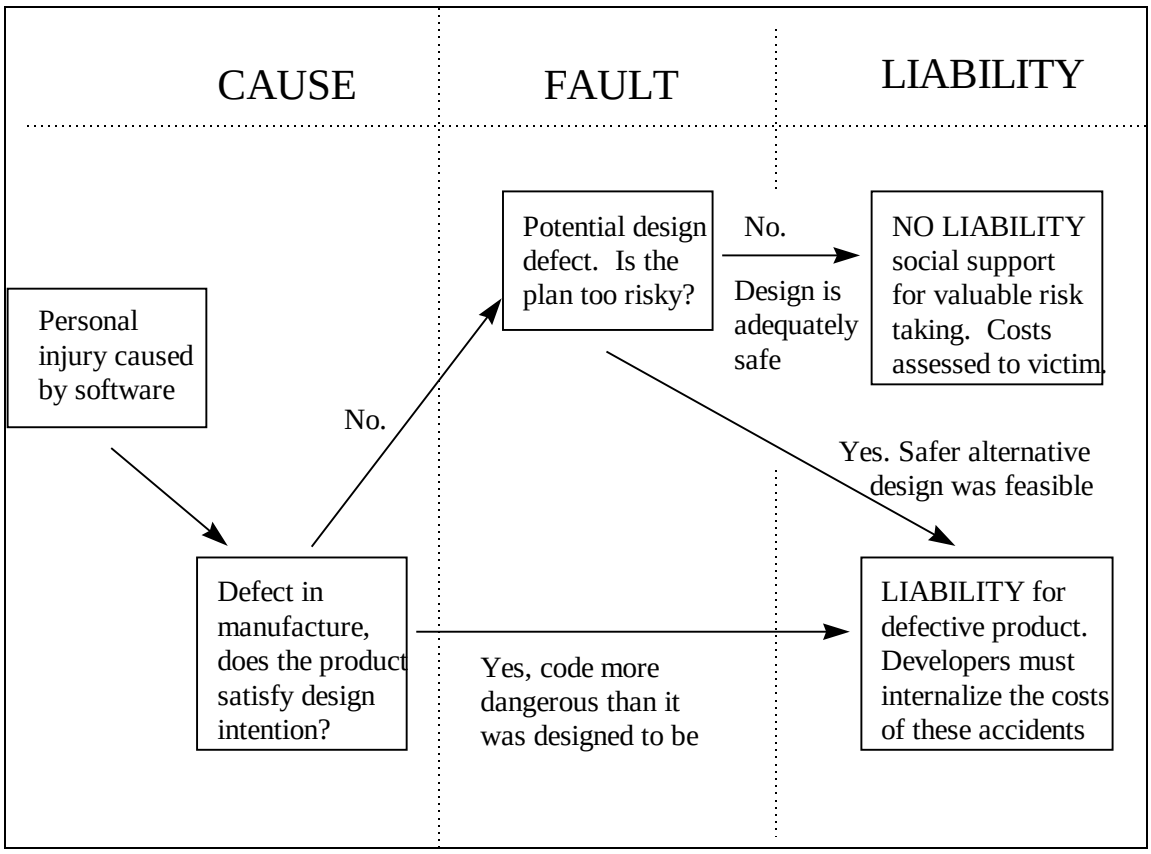

Figure 4 - Anatomy of a Case

Once software is proved to have caused or contributed to a personal injury, either kind of defect may be shown. The burden of proof and expected costs of prosecuting a case motivate the plaintiff to allege a defect of manufacture. This is the first issue to be decided, and if a manufacturing defect is identified, the plaintiff wins the

lawsuit.[12] If the alleged defect is not one of manufacture, then defective design is at issue. Expert testimony can now be introduced by the defendant-designer in defense of the reasonability of the design. This defense was not available in the manufacturing case. The determination of liability is then made respecting the much more complex issues of design and state of the art.

Determination of Defect Class

As shown in the *Restatement Third*, Courts seek *intended design* as the marker to determine whether there is a manufacturing defect. Where is this intended design obtained for use by the Court in a given case? It can be that the product "deviated in a material way from the manufacturer's specifications or from otherwise identical units manufactured to the same manufacturing specifications." [Rix86] This reveals the two basic ways that Courts actually determine this marker:

1. design specifications; and,
2. deviation from the norm.[13]

There is one other way to prove a product defective, that is, when it fails to perform its "manifestly intended function." [Res98] This is reserved for cases in which the public has a wealth of basic experience and knowledge with the product at issue, and expert testimony is not required because the knowledge is socially accepted as

[12] In some cases, there are good reasons to go on to prove design defect in addition to manufacturing defect, such as the availability of punitive damages. Indeed, both classes of defect may be alleged and present in a product. [Pro84]
[13] This test for manufacturing defects was so named by Justice Traynor. [Tra65]

universal. This is not a likely scenario for a software product for some time to come, if ever. [Nyc79]

1. Design specifications as an expression of intended design

It is important to understand the broad usage of the word "design" as used in the products liability context. From the viewpoint of the remote customer in society, any choices of the manufacturer that involve foreseeable consequences for safety would be included.

When the Court searches for the manufacturer's intended design, a natural starting point is internal design documentation for the product. After all, the manufacturer often uses such documentation in its own construction and quality control efforts. In this sense, these documents are produced precisely to exhibit the manufacturer's intention: a definition of what the manufacturer intended to produce.

The utility of a comparison to design specification documents depends on the existence of a complete, consistent, correct, unambiguous, comprehensible expression of the product design. It is to no avail if the documents are no longer available, if product features are not traceable to their counterparts in the specification, or if the specification is otherwise insufficient to answer the question, "is the specification satisfied?"

2. Intended design in the manufacturing norm

If the specification is missing, if it is not comprehensible, if it is incorrect, inconsistent or ambiguous, it cannot reliably be used to divine "intended design" so that the individual product feature may be evaluated against it. Since the Court needs to classify the potential defect in all cases that come to it, some other way is needed to determine whether a manufacturing defect exists. The "deviation from the norm" test compares the product in question to a number of others from the same production run. [Tra65] If the given product defect is found in all the others, then a defect in design is inferred. This fact gives rise to the descriptive term *generic defects*, referring to those defects that affect the entire product line (by design). [Res98] If the product feature in question does not appear in the majority of others, then it is inferred to be one of manufacturing: an unintentional failure to execute the design properly for that individual product.

This test is easy to understand and its genius lies in the ability to infer intended design from some sample of products - without reference to the design specifications at all. Simple and practical, it circumvents many known problems of a test to the design specifications.

Summary of Important Points

Given a complaint from an injured party, products liability in tort must determine what kind of software defect is at issue and assess monetary responsibility based on

that classification. For engineering design decisions that involve risk, a potential defect in design is at issue. Whether a defect exists in this case is determined by a social standard for acceptability of risk. For inadvertent errors in constructing the product to the objective standard of the manufacturer's own designs, a manufacturing defect is found. There is no defense of due care to this sort of defect.

The distinctions between these two classes of defect are important to society as well as to the designers (and injured parties). It is imperative that the law have a deterministic algorithm to decide into which class a potential defect falls.

Chapter Four:

Constructing Software Code

The terms “defect,” “design,” and “design specifications” can be used in describing the software engineering process but have already been used in the legal sense in the previous chapter. This chapter will substitute the term “flaw” for “defect” and will describe, in the generic sense, anything found lacking in the software process or product. The term “design” has no simple substitute and will be used in the software engineering sense except if otherwise noted. The term “specification” is often used interchangeably with “design specification” and “requirements specification” and that broad usage will suffice for the purposes of this chapter.

The Kinds of Software Considered Here

The analysis of this dissertation may not apply to all kinds of software. In particular, the software considered here has the following attributes:

1. nontrivial in size and complexity;
2. reaches or affects a mass market of indirect “customers;”
3. is produced in a human readable (compiled) programming language; and,
4. has the potential to contribute directly to risk of personal injury as a part of a physical system.

Examples of this kind of software include automated nuclear plant shutdown systems, medical linear accelerator systems, and automobile antilock braking systems. Medical expert systems, one of a kind simulators or unique experimental systems would not be considered here.

Constructing the Software Product

Production of the software product involves, at the very least, coding. Coding activity has been variously described in the literature. It is often referred to as "construction" [GG75] [Som92] by those in the field. It has also been referred to as "translation" of design into code [Dav95], where code activities are seen as a clerical task [Gem81] that might not encompass any engineering judgment which might affect product safety. However, if coding really is mere "construction" that does not require engineering judgment, then it would certainly be fully automatable. Though there has been some success in very small domains with simple problems, it does not, in general, affect software development as a human activity. [Bro95] Therefore, when coders produce code from a software design, some kind of expertise is thought to be required. But then, what is the difference between producing source code (coding) and code design? Let us first review the definitions of some relevant terms from the Institute of Electrical and Electronics Engineers (IEEE).

"Source code" is defined in the IEEE Standard Glossary of Software Engineering Terminology (1994 Edition) [IEE94] as:

> **source code.** (1) Computer instructions and data definitions expressed in a form suitable for input to an assembler, compiler, or other translator.

The definition of “coding” is given as,

> **coding.** (2) The transforming of logic and data from design specifications (design descriptions) into a programming language.

Thus, construction of source code is seen as the “transforming” of design information from design specifications into source code (a programming language), which is suitable for input to some mechanical translator to translate into machine readable form. "Design" is defined,

> **design.** (1) The process of defining the architecture, components, interfaces, and other characteristics of a system of component.
> (2) The result of the process in (1).

Software engineering textbooks generally explain that “detailed design” is the form of the design used to provide the coders their work assignments. [Sch99] "Detailed design" is defined,

> **detailed design.** (1) The process of refining and expanding the preliminary design of a system or component to the extent that the design is sufficiently complete to be implemented.
> (2) The result of the process in (1).

These definitions do not give a very complete picture of what the coder is expected to do or what a flaw in coding might look like. The definition of coding is “translation of ... design specifications... into a programming language.” “Design” is

defined in terms such as “components.” The salient characteristic of detailed design is, “sufficiently complete to be implemented.” The definitions of code and detailed design reference each other and are circular. This is an example of the “chaos” that reflects the state of the field. [Jac95] There is no clear definitional boundary found to distinguish these two activities.

Clearly, though, software that has the potential of causing or contributing to personal injury is not constructed in a haphazard manner. The discipline of “software engineering” has arisen to provide “the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures and associated documentation.” [Boe81] A careful software development process is required to produce safety critical software products. Study of the process will give insight into the nature of code construction and the flaws that can result there.

The Software Development Process

The purpose of the software development process is to provide a software solution to some problem. Customer requirements are gathered, design plans for a solution in software are developed, and software code is written. The code is then mechanically compiled from its human readable form into a machine readable form. The machine

readable, or executable code can then be copied identically[14] and marketed to the public.

Notice that the software development process is a human effort geared towards *producing one code product* for a given release.[15] The copying process may produce *n* copies for the market, but they are the same product. This fact is illustrated in figure 5.

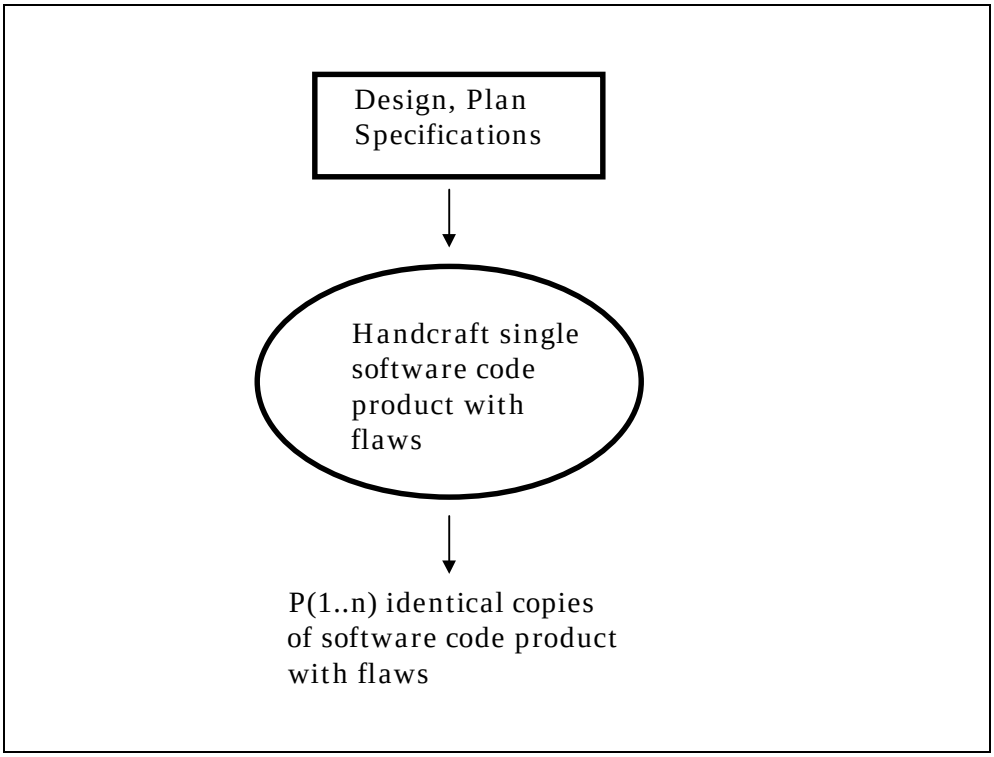

Figure 5 - Software Production

[14] Notice that flaws in the copying process are possible, but the chances are so insignificant that they will not be considered here. See generally [Par90] or [Ham92] for more details and further explanation.

[15] [Par90] references regression testing and explains that any changes to the code result in a completely new software product.

What is the nature of the flaws that can creep into this single software code product? Recall that the law of products liability is generally concerned with classification of flaws that exhibit the software engineer's planning (design) and also with failures to produce a product that implements that plan correctly (manufacture). To further investigate the potential for code flaws of this sort, the software development process is reviewed in more detail.

Code is the Product of a Human Effort

Code production has not been automated to any great extent. [Bro95] It is still the result of an intense human effort involving, at the very least, interpretation of specifications and code writing to satisfy them. [Par86] notes that, "human errors can be avoided only if we avoid the use of humans." Humans are capable of a range of simple "mistakes." Two that are relevant to this work are listed below:

- flaws in the coder's interpretation of a specification; and,
- flaws of transcription or simple mistakes in coding accuracy.

If these are the only kinds of flaws that appear in code, then coding is a simple clerical task to be performed purely as a function of the given specification. A review of software engineering thinking about the process leading to coding is instructive in this regard.

Process Models

Code and Fix

Early efforts to produce code were based on simpler tasks than are attempted today. A carefully designed process of specification before producing code just wasn't necessary for the simple tasks attempted. The basis for code production was simple: a single coder considered the problem and attempted to use a program to solve it. The first attempt was expected to fail, so the code would be modified in the attempt to revise the code solution to make it workable. In this way, a piece of working code might eventually be produced, and "code and fix" is the process, shown in Figure 6.

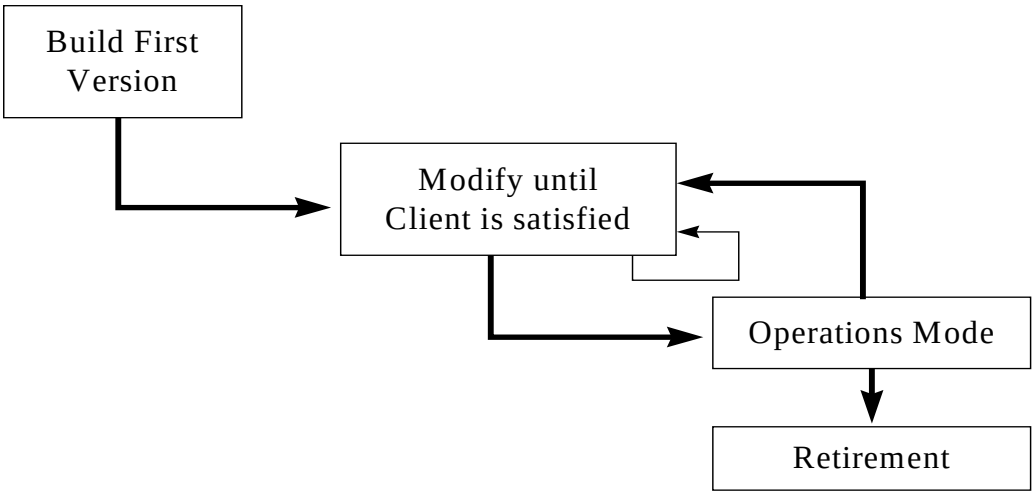

Figure 6 - Code and Fix

A specification is not necessarily produced when developing code this way. The customer's requirements reside in the head of the coder, where the design is

created. The design is then written and constructed as a code product in one fell swoop. Not a bad idea for trivial coding tasks where size, reliability, safety or maintainability are not serious issues.

As projects grew in complexity and size, other nonfunctional requirements such as reliability, safety and maintainability became more important issues. These issues could not be addressed effectively from the code and fix perspective. [Sch99] Careful, systematic planning was used to help in development of nontrivial projects where higher quality was desired.

Waterfall model

In the 1960's and 70's, software developers began to think of the software development process for any nontrivial project as a sequential or "waterfall" model. [Bro95] The process "stages" of planning, coding and testing are formally separated in the attempt to break down a complex process into smaller, more manageable subtasks. Testing to detect flaws is performed on the code, where flaws may be corrected until the code product is of satisfactory quality. This process is illustrated in figure 7.

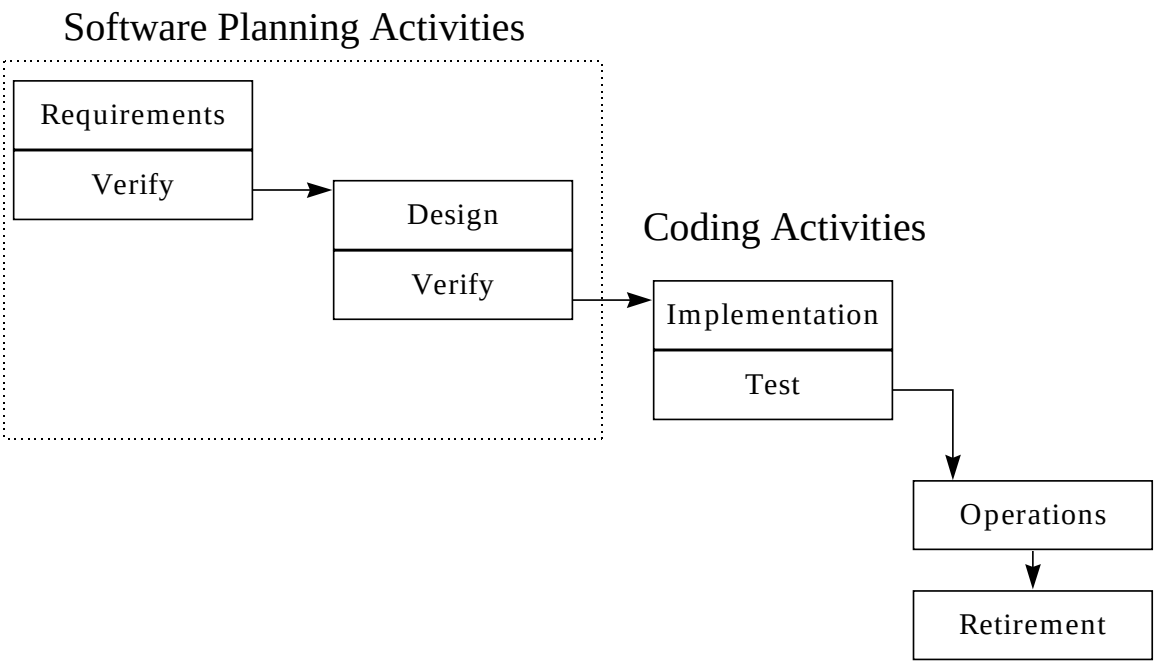

Figure 7 - Waterfall Model

Notice the following things about this model:

- the project goes through the process once, entirely “top-down” - all design decisions are made in the discrete planning stages, before code is constructed;
- code is the “construction” activity following design, its purpose is to construct code to satisfy design intention; and,
- all flaws reside in the code, where the testing is performed to detect them.

In 1970, Winton Royce [Roy70] observed that this model did not suffice to describe reality, so he proposed the addition of feedback loops. This feedback is limited to an immediately preceding stage in order to limit process complexity. The more realistic model is shown in figure 8.

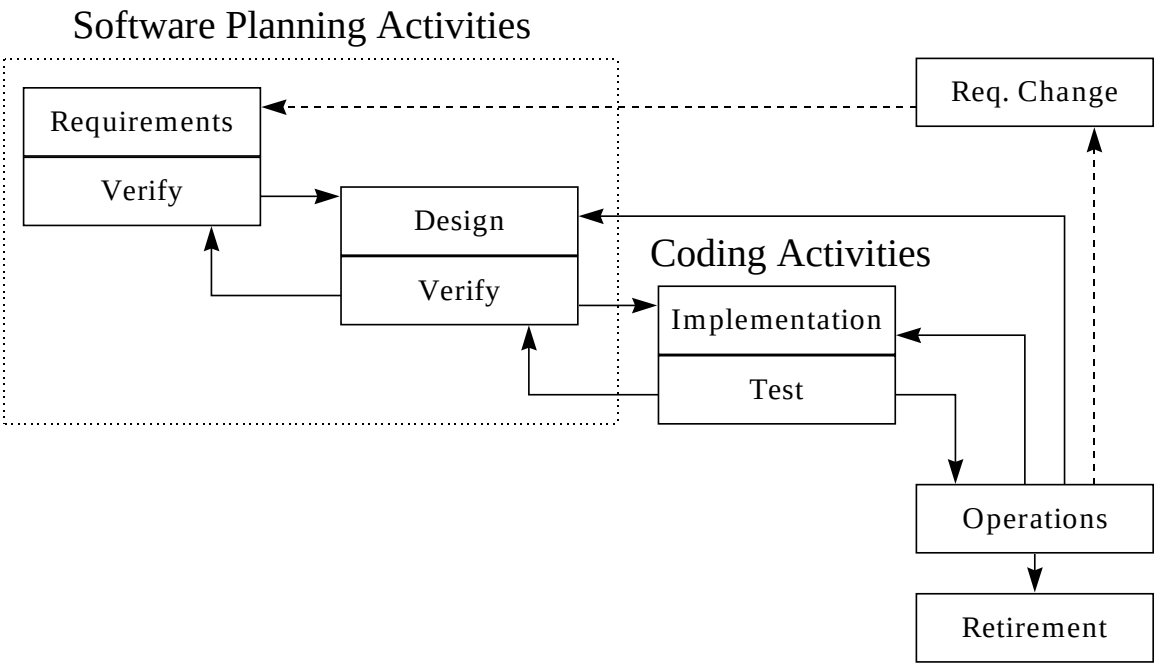

Figure 8 - Waterfall with Feedback

While this was recognized as an improvement, it was not really surprising. It is true for many mass produced mechanical devices. The actual construction is likely to result in some lessons for design. [Pet92] For software developers, this new model was an explicit recognition of the fallacy that software may be developed systematically through consecutive, discrete stages of development. Feedback from the coding to design is necessary for any real project. But this implies that specifications are not the self contained activity originally envisioned.

Inevitable Intertwining

[Bro95] explains that, "[l]ike the energetic salmon [...] experience and ideas from each downstream part of the construction process must leap upstream, sometimes more than one stage, and affect the upstream activity." [Bro95 at 266] [SB82] goes

further and explains that *specifications can never be self contained*, that specifications and implementations are "inevitably intertwined." They base this conclusion on two factors: design realities discoverable only during construction of code and imperfect foresight during the specification activity. In their view, implementation is a process of "specification modification."

[Par86] in basic agreement with these views, attacks the idea that a "rational design process" could ever exist for software. He focuses on inherent inadequacy of the software specifications as the basis of his argument. He finds that specifications will never be adequate to the task of completely specifying the code for the following reasons:

- incomplete information about original product requirements;
- many design details only become known as construction proceeds, designers cannot foresee all consequences of their decisions before construction;
- for complex products, humans are unable to fully comprehend all the details to design and build the correct system;
- project specifications are subject to change for external reasons;
- human errors are expected in a human activity like specification;
- preconceived design ideas involve less than ideal designs; and,
- economic pressures for reuse of less than ideal components.

These factors result in specifications that are less than perfect, and will be incorrect, ambiguous, inconsistent, or incomplete in some measure. [Par86]

The specification insufficiency problem is really quite difficult, even for seemingly small and simple sets of specifications. An example is given in [Mey85] of a simple specification given by Naur in 1969 that was "proved correct." Goodenough and Gerhart [GG77] showed seven errors found in Naur's specification. In 1985, [Mey85] finds several other specification problems with the "corrected" version. One might wait for the next paper to come along.

Spiral Model

The currently accepted model of the software development process is a generalization of many of the previous models. [Boe88] This spiral model explicitly recognizes an incremental backstepping through the process in a spiral that could be infinite in length. Boehm admits that specifications produced are not necessarily uniform or complete during the process and that incremental efforts to increase their sufficiency are based on risk priorities and resources available. Figure 9 shows the spiral model.

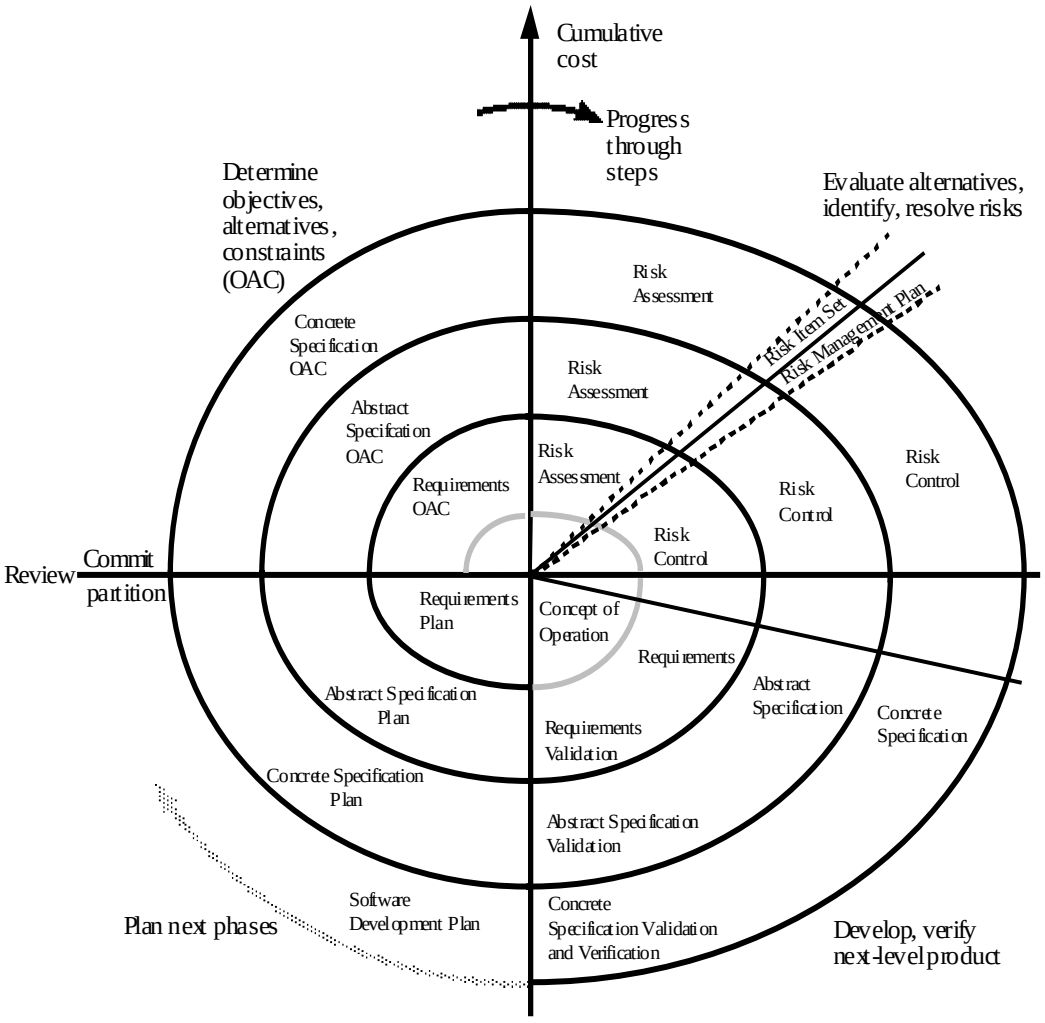

Figure 9 - Spiral Model

Coders Must Face Design Issues

Coders face the weight of the process bearing down on them in difficult situations. They are at the “end” of the code product effort in time and resources. The pressure is on them to produce a working product, to modify the product when needed, to

repair any flaws that have been found. Coders are expected to deliver the working product for final distribution and sale. [Sch96] They suffer this pressure within the framework of market pressures: limited resources and time deadlines. [Bro95]
If the specifications are inevitably insufficient to provide all the coders need to do an adequate job, what is their option? They *must perform design activities during code.* These two possible cases illustrate their dilemmas:

- "no specifications;" and,
- "insufficient specifications"

In the case of no specification, it is clear that coders must engage in all the activities needed to produce a working code product. It is common, even for safety-critical software products, to find that there is no separate specification document in existence.[16] The software code itself sufficed for conceptualization of design and implementation entirely. An example of a safety-critical software system apparently implemented without any software specification separate from code is the Therac-25 linear accelerator built by Atomic Energy of Canada, Limited. [LT93]

Even when there are specifications, even when they are carefully produced and analyzed, they are not expected to be perfect or self contained. [Jaf88] They certainly cannot ever be proved to be so. [Lev95] They will not contain all the information necessary to code up a solution to the problem at hand. Common forms of insufficiency for specifications include incompleteness, incorrectness,

inconsistency and ambiguity. Each of these has direct consequences for the coders, under pressure to build a working product.

Software specifications that are not "complete" cannot be counted on to express full design intention. They are only a partial expression. Coders faced with incompleteness may need to fill in the gaps for design, similarly to the no specification case. Incorrect specifications cannot be counted on to express true design intention. Coders faced with this problem may need to make the "correct" decisions if they are discovered late in the process or are subtle in nature. Inconsistent specifications can exhibit opposing design decisions depending on what part of the specification is consulted. Coders, if faced with this problem, must choose some interpretation and build around that. Ambiguous specifications can be interpreted to yield different designs with equal authority. Coders may need to resolve ambiguities in order to proceed building a working code product.

These problems are faced by coders everyday. Real world systems have been developed using specifications known to be incomplete. For instance, the TCAS system was developed entirely from a specification where much of the design intention was missing. [LHH94]

[16] This information comes from private communications with several coders working on safety critical and other software systems.

Summary of Important Points

In this chapter, it has been shown that software is produced such that every flaw produced in the code is expected to appear in every copy of the product released to the market. It has also been shown that definitions of “code” and “design” are in a state of confusion within the field of software engineering. Specifications are found to be insufficient to enable the coders to perform their jobs in a clerical manner. Code construction is found to be an imperfect process: the idea that software “design” occurs entirely separately from, and previous to, software “construction” is an unrealistic idealization.

Chapter Five:

Defects in Software Source Code

The possibility of legally cognizable product defects is now investigated for the software code product. First, the law must be shown applicable to software. Next, simple attempts to externalize the costs of such liability are dismissed. Finally, code flaws are demonstrated that map into both classes of defect: manufacture and design. Extant tests to distinguish the defects are applied and shown insufficient to decide.

Is Software Subject to the Law of Products Liability?

Recall from chapter 3 that a products liability case needs to involve the following:

- personal injury or property damage (not pure economic injury); and,
- the instrumentality causing the harm must be considered a "product" for purposes of products liability.

Personal Injury

The first prerequisite is satisfied by definition: software as considered here (see chapter 3) can cause or contribute to personal injuries and is not restricted to economic damages. This sort of software is increasingly prevalent in products

produced today, injuries have already occurred and no technical cure is expected. [LT93]

Software Code as a "Product" or "Component"

The second prerequisite is a question of law to be decided by the Courts. The factors to be considered include: whether software code has a value of its own as a salable item, can be "owned," is subject to correction and modification, and whether it is mass produced and marketed. [Mor89] The sort of software considered here has value of its own that is embodied in some medium for sale and ownership. It can be mass produced and reaches a mass market of remote customers. Many legal commentators argue on the basis above and other bases that safety critical software should be considered a product for purposes of products liability in tort. [Gem81] One court has argued in dictum that, "software that fails to yield the result for which it was designed may be" an example of a product subject to products liability.[17] [Win91] This sort of dictum, though not binding, is often seen as a signal from Courts as to the eventual outcome on the issue. [NLJ91 at 3]

One legal commentator has further argued that software may be considered subject to products liability as a "component" of the computer regardless of its individual status as a product. [Gem81]

---

[17] This court also addressed the issue about whether the artifact has any dangerous propensities by itself. Software, of course, does not, it cannot hurt anyone by itself. The court argued about aeronautical charts that, "[A]lthough a sheet of paper might not be dangerous, per se, it would be difficult indeed to conceive of a salable commodity with more inherent lethal potential than an aid to aircraft navigation that, contrary to its own design standards, fails to list the highest land mass immediately surrounding a landing site." The court cites [Flu85].

Overall, there are very few legal arguments against the application of products liability in tort. Most authorities consider the issue settled [Wol93], though legal precedent has yet to be set.

Can the Costs of Products Liability be Easily Avoided?

Many software developers carefully craft special disclaimers in an attempt to protect themselves from products liability. But as noted in chapter 3, the law states that liability for product defects cannot be defeated by contract or license disclaimers. [Res98] Developers may choose to protect themselves from liability costs by insurance. Such insurance may become available with increasing demand. Insurance rates are, of course, subject to an assessment of the liability risks of the insured. Thus, rates will eventually reflect the estimated costs, spread over the industry. Liability is therefore still an important consideration for software designers. There is no easy way to avoid the costs.

Software has been shown to be subject to the law of products liability. There is no simple way to avoid the costs of liability. The nature of liability and expected costs of liability must be investigated.

Research Question

So far, this dissertation has explained the basics of products liability law and the nature of software code construction. With this background, the research question may now be asked:

> **Given a flaw in software code to which a personal injury may be causally traced, can that flaw be reliably classified as a defect in manufacture or design?**

For purposes of discussing this question, make the following assumptions:

- mass marketed software is involved;
- personal injury is involved;
- the injury is causally traced to small fragment of code for investigation;[18]
- the plaintiff (injured party) files a products liability lawsuit alleging a defect in code manufacture; (most cost effective for this party if provable)
- the defendants (designers) claim the issue is one of design and that the lack of cost effective safer alternatives exonerates them from liability; (due care exercised)

This hypothetical will provide the basis for the analysis of this chapter. The basic argument of the chapter is that both classes of defect are possible in software source code. A court must necessarily be able to distinguish one class of defect from the

[18] There is a whole range of causation problems that might be discussed at this point. However, such a discussion is beyond the scope of this dissertation. The interested reader is referred to [Bei96] for a discussion of causation and software bugs.

other. However, the available tests are shown to be insufficient to the task in a whole class of realistic cases.

Classes of Defects that Originate in Code

Some in the legal and software fields claim that software (code) is pure design, but they make the claims without any legal analysis of the implications. If they are correct, and the only defects possible in software code are considered defects in design, then legal responsibility for defects is defined by a single standard: negligence.[19] Though some work would remain to be done to define the "due care" of a software engineer in producing code, the standard of responsibility is well understood and could be applied. [Kan95] [TRK96]

Design

There is no argument in the literature about whether design defects can appear in code. It is clear that design decisions made at a higher level of specification could involve poor analysis of social risks and utility. Such decisions could well result in a judgment of design defect. The code product that meets such specifications contains a defect inherited (correctly!) from the higher level.

[19] Legal concern for inadvertent code defects would necessarily turn to what has been called "inadvertent design defects." [Hen76] But a determination of an objective standard is required by which to judge such defects. Essentially, this results in the exact same problem analyzed here.

Recall from chapter 4 that some design decisionmaking goes on right in the code and is not always covered by a separate specification. This sort of design decision is "native" to the code. It proves to be more troublesome than the inherited design decisions.

A short list of possible "native design decisions" (not intended to be anywhere near exhaustive) include:

- choice to manipulate global or local data, problems related to scoping;
- choice of particular data structures to represent data;
- choice of algorithm construct such as "do-while" or "repeat-until;"
- choice of ways to implement exception handling; and,
- choice of communication mechanisms.

Each of these decisions can have safety implications under the right circumstances. For instance, the choice of global variables to represent shared data between concurrent processes may be made on the basis of memory efficiency and speed considerations. However, the problems of deadlock, livelock and starvation may occur if a nontrivial mathematical solution is not correctly designed and implemented in the code. [Jac95] This very problem was found in the Therac-25 code and proved responsible for several horrible accidents. [LT93]

Manufacture

If software source code is shown to exhibit defects that can be legally characterized as defects in manufacture as well as defects in design, then the ability to distinguish them in a reliable manner is at issue. The legal standards are different and this difference will be quite significant to software engineers in their risk management efforts.

In chapter 3, fundamental characteristics of defects in manufacture are contrasted with fundamental characteristics of defects in design. Recall, as shown in figure 3, fundamental characteristics of the defect in manufacture include:

1. the defect itself is discovered by comparison to the internal design standard of the manufacturer;
2. the essence of the defect is inadvertence, not inadequate engineering intention;
3. the danger due to the defect is not avoidable by improved engineering analysis; and,
4. the defect is not known beforehand, it is latent (not detected in test).

There is no argument in the literature about the nature of a physical defect in a diskette or other physical media: it is a product manufacturing defect in the traditional sense. There is some argument in the literature, as seen in chapter two, about whether defects in manufacture will apply to code at all. This question must be settled.

An Example

Consider the following simple example:

Specification: *Increment ThisVariable*

Code: *X := X * 1;* */ Notice the mistake, should be "X := X + 1;" /*

This flaw is an unintentional failure, coders make such mistakes because they are human. The flaw is certainly latent, else the product would have been rejected by the manufacturer's quality control or testing group until it was corrected. This flaw shares all the basic characteristics of a manufacturing defect. Does it fit the formal legal definition?

Recall the salient part of the *Restatement* definition:

> **... when *the product departs from its intended design* even though all possible care was exercised in the preparation and marketing of the product; ...** (emphasis is mine).

The essence of this definition is a departure from intended design. The failure of the coder to correctly satisfy the software design specification exhibits a "departure from intended design" by definition. The essence of the defect is the difference between the designer's intention and the coder's construction of the code product to satisfy that design intention. This defect is clearly an undiscovered mistake in

implementation of the design intention. Thus, the software product contains a defect in manufacture.

Such flaws are not uncommon in software code. Some may be eliminated by the testing and analysis process. For instance, simple syntax flaws are often caught by compilers. Some flaws may be caught during specification based testing. [RAO92] However, testing can only reveal the presence of flaws, not their absence. Some testing is designed specifically to reveal particular classes of faults. [RT88] However, *testing cannot reveal all such flaws in the code product.* Inevitably, many flaws remain in code that is released. [Par90] Some typical examples of ways to produce code that inadvertently fails to satisfy design intention are:

- inadvertent operator, numerical, variable substitutions in expressions;
- inadvertent permutations of elements of a noncommutative expression;
- inadvertent factoring of nonassociative terms in an expression;
- inadvertent "cut and paste" editing, variable scoping problems; and,
- inadvertent errors in complex logic expressions in conditionals.

This list is by no means exhaustive, but meant to illustrate the broad possibilities for manufacturing defects in code due to coder inadvertence. All of these kinds of defects have the potential to cause unexpected paths through the program's state

space with unexpected consequences for safety.[20] Notice that analysis of requirements and designs have no ability to detect or eliminate such defects. These defects are the byproduct of a human construction effort.

Overall, though code can contain design flaws, it is not "pure design" in the legal sense. It has the potential to contain defects in manufacture. The ability to distinguish the classes of defect is now critical to the proper operation of the law of products liability.

The Classification of Software Code Defects

The foregoing has shown that flaws in software source code have the potential to be defects in design or defects in manufacture. The difference in social and liability costs for these defects is significant, and thus they must be distinguished. As explained in chapter three, the common law of products liability has two ways to distinguish these product defects from one another:

1. the deviation from the norm test; and,
2. a test of comparison to the manufacturer's "design specifications."

[20] Work has been done to find conditions where code faults will cause erroneous output and alert testers that a problem exists. [RT88] Other work is in progress to develop a notion of software's *sensitivity* or *tolerance* to such defects. [Voa97] This work aims to quantify risks of software hazards due to specific kinds of defects.

1. Does the "Deviation from the Norm" Test Suffice to Distinguish Code Defects?

The deviation from the norm test derives design intention from the production norm. However, the software code product is built only once in release version and exact copies produced. There is thus no "deviation" from the "norm" - the one product produced for release. In general, each and every defect of either class will appear in each and every copy of the software product to reach the market. Comparison of the purportedly defective product against others like it will reveal no deviations at all. This test fails to distinguish the classes of defect.

Notice that the foregoing conclusion does not apply to defects in software configurations, where the possibility exists that a comparison to other similar products can reveal a difference and expose a defect. This is not the general case and will not be further developed here.[21]

2. Does a Test Against "Design Specifications" Suffice to Distinguish Code Defects?

Since the deviation from the norm test fails, a test of comparison to design specifications is the only alternative.

Application of this test involves a comparison of the product as constructed against the design specifications as an embodiment of the engineer's design intention. The

[21] One might also imagine that n-version programming might fall within these tests for defects, but since the code may be written differently from version to version, manufacturing defects are likely not discoverable by this test.

last chapter used a broad definition of “specification” that is applicable here. Under the law of products liability, design involves any decisions that foreseeably affect product safety. Thus, “design specifications” may include decisions made in requirements on behalf of the consumer, software design, and may even encompass comments in code.

If the potential defect is shown to fail to satisfy the design specifications, then the defect is classed as one of manufacture, a mistake in construction of the product. If the potential defect in source code is shown to have properly satisfied design specifications, then the potential defect is classed as design and a risk-benefit analysis must ensue to determine whether it really rises to the level of a product defect.

This test is broken into two separate cases for consideration:

a. no specification exists; and,

b. some specification exists.

a. No Specification Exists

This case is actually quite common, even in the field of safety-critical software.[22] The Therac-25 linear accelerator code is an example. Consider the following fragment of code in figure 10.

[22] Private communications with coders who work on such products.

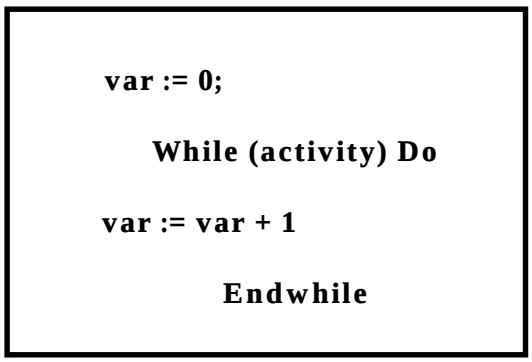

Figure 10 - Therac Code Fragment

The code maintained an 8 bit global variable as an *apparent* indicator to other code that a certain activity was taking place. The variable had been initialized to zero before the activity began. During the activity, the variable was incremented and it held a positive value while the activity took place, except during once cycle when the value rolled over to zero (after 256 cycles). At least one injury has been traced to the reading of a zero from this variable while the activity was still taking place. [LT93]

This certainly could be a defect in design. The coder's judgment about the maximum number of update cycles during the activity might be at issue. The cycle count itself may have been a design issue from testing or other analysis.

On the other hand, it could have been a mistake in implementation. The coder may have meant to write "**var := 1"** instead of "**var := var + 1"**. The initialization of the var to zero may also be in the wrong scope because of a simple cut and paste error. These would yield a defect in manufacture.

When there is no specification separate from the code, there is no objective evidence of intended design for comparison. A Court has no way to decide the question.

b. Some Specification Exists

As discussed in chapter 4, software design specifications have been shown to be inherently incomplete, inconsistent, ambiguous, and incorrect to some degree. They do not fully and reliably embody the intention of the software engineers who designed the source code. Similarly to the "no specification" case, if there is no objective evidence of the designer's intention, the Court cannot decide what kind of defect faces it. Thus, the test of comparison to design specifications is not sufficient to classify software source code defects.

This difficulty has been experienced in the current airborne collision avoidance system (TCAS). [LHH94] wrote about the specification that "the intent was missing. Therefore, distinguishing between requirements and artifacts of the implementation was not possible in all cases." [LHH94 at 705] This was in the context of an effort to "reverse engineer" formal specifications from legacy pseudocode specifications. Insufficiency remains as the specifications are continually maintained.[23]

[23] The benefits and limitations of such an approach are evaluated in the next chapter as "post hoc rationalization."

In conclusion, though these tests *may* work in some cases,[24] they do not work in some real world cases. Given an arbitrary code fragment to examine, as will occur in a court case, the extant tests cannot be counted on to distinguish the class of the potential defect.

Summary of Important Points

In this chapter, it has been shown that software code can harbor flaws of engineering judgment (design) or plain failures to implement the design correctly in the code itself (manufacturing). However, both given tests for distinguishing these classes of flaws fail in known situations.

[24] There is reason to believe that even when the test "works," it may not be revealing information useful to application of the primary social goals behind the law. This is discussed further in chapter 6.

Chapter Six:

Specification Insufficiency: Essence or Accident?

To this point, it has been shown that software will be subject to products liability in tort for accidents involving personal injury. Defects of both different classes may be present in the code. Distinctions between these defect classes is crucial to the rational operation of the law. Two methods have been developed to distinguish these defects for products, and these methods have sufficed for all products to come before the law so far. These methods do not suffice to distinguish defects in the software product.

This chapter will look at the impediments to classifying defects in the software product to see if any promising approaches exist. The deviation from the norm test is shown nonadaptable due to the basic nature of software production. The possibility that progress will improve the sufficiency of specifications (to distinguish design from implementation) remains. This possibility for improvement does not hold a solution for two essential reasons:

- progress towards a solution to the software specification problem is limited; and,
- solution to the technical problem of software specification does not solve the legal problem of classification of defects.

First, the non-adaptability of the deviation from the norm test is explained in detail. Next, the possibilities for technical progress in software specifications are reviewed. Finally, fundamental difficulties with *any* technical solution are demonstrated. This difficulty is shown to be an *essential* difficulty in the Brooks sense, it will not yield to simple solutions.

Can the "Deviation from the Norm" Test be Adapted to Software?

For traditionally manufactured products such as lawnmowers and automobiles, the individual products are essentially constructed to meet design specifications one at a time, individually. Random flaws appear and are distributed among the individuals of the production run. Thus, manufacturing defects themselves are *not* generic in nature. Design defects, on the other hand, appear in the entire product line uniformly, so they are often referred to as "generic." [Res98]

For software, a single code product is constructed to meet the design specification and then identical copies created. This has drastically different consequences for the distribution of manufacturing defects over the number of products sold on the market. Manufacturing defects may be randomly distributed, but over the structure of a single product! These defects then appear identically in each and every copy of the product. Thus, *all* defects of software code construction may be

termed “generic.” This defines a new class of product defect: the *generic manufacturing defect*.[25]

Clearly, there is no way to modify the deviation from the norm test to work for the software product. As seen in chapter 4, identical copies are created for the market. The norm for the software product line is the single release version. There are no deviations to distinguish defect classes.

Since the deviation from the norm test cannot be adapted to the software product, the sufficiency of software specifications must be reviewed as the only potential solution to the problem.

Can Software Specifications be Made Sufficient to Distinguish Defect Classes?

Two arguments will be advanced that software specification can never be entirely adequate, in general, to classify arbitrary defects in software code. The first argument is based on limited resources and the ultimate difficulty of the task. Since the law needs a deterministic algorithm (that halts) to decide the defect class of an arbitrary flaw, partial solutions won’t work.

The second argument is more fundamental, and is based on the essential nature of software code as a product. The medium of design and medium of implementation

[25] So named in a conversation with Cem Kaner about these issues.

for code have no objectively observable boundary. Attempts to draw lines are shown to be subjective. Distinctions drawn from such boundaries are therefore not sufficient for legal decisionmaking.

Specification Insufficiency and Progress in Software Engineering

Software engineers recognize that specification completeness, consistency, correctness and ambiguity are serious problems. [Jaf88] Progress is important and active research is conducted into improving the situation. Several areas are reviewed briefly for their potential to solve the specification problem: post-hoc rationalization, software design standards, and formal specifications. This is a short list, but represents a range of views and exposes some basic limitations on technical progress in software specifications.

a. Post-hoc rationalization

[SB82] suggest that a distinction between specifications and their implementations "can only be made after the fact." [SB82 at 439] Ideally, all design decisions made (or changed) in code could be recorded in the design specification after the code is fully constructed. In the end, this would result in a more complete, consistent, unambiguous and correct set of design documents. [Par86] expands on this idea and derives a post-hoc rationalization of the entire process where the design specification would be updated to a rational state even though the process could not have worked that way in reality.

Substantial factors militate against creation of an ideal document capturing full design intention [and fully distinguishing design from the code]:

1. Limited resources
    - expensive to devote the time and effort to full post-hoc rationalization;
    - only possible *after* the product is ready for release but market pressures dictate release without delay.
2. Inherent difficulty
    - subtle design knowledge is difficult to communicate to others;
    - notions of what belongs in the design specification are not clear;

The first reason is based on limited resources. The task of producing ideal documents would require a serious commitment of resources to a nearly perpetual task. Market pressures on developers dependent on profits for survival make long delays in release nearly impossible.

The second reason is the pure difficulty of the effort to produce ideal documentation, even after the product is produced. The problem of communicating the design intention to others on a team, even when there is a systematic effort, is a difficult one. [LHH94] Not only that, but those working on the problem have no clear definition of what belongs in the "design" document distinguished from what belongs in "code." For these reasons, it is unlikely that post hoc rationalization can ever meet the goal of sufficient specifications.

b. Design Standards

Jackson notes that software engineering does not have the advantage of a narrow focus, that software engineers are really analogous to "physical engineers," "imaginary polymaths who understand how to specify, design, and build any useful physical object whatsoever, in any material, to serve any purpose." [Jac95 at 189] The only real successes in software engineering have come in the specialized branches, such as compilers, databases, and operating systems. There are no standardized designs for software in general. [Lev95] Software may not even be amenable to such standardization. For engineers in the established branches of engineering, they don't focus on the problem to be solved as much as they focus on problems of a relatively small and well-defined class. "The design solutions are already well classified, and with them the problems that they solve." [Jac95 at 189] But software is not often like that, each software system to be designed is not likely a different solution to a standard problem, but a solution to a different problem.

Though software specification may not be amenable to general design standards, progress may be possible by narrowing the design space within which software engineers devise solutions to closely related families of problems. This is one of the implied goals of research in software architecture. A software architecture is a particular form of software design, [GS93] and it serves primarily as the "big picture" of the system under development. [RM98] Architecture Description Languages (ADLs) and architectural styles provide a framework for modeling a software system's conceptual architecture and thus may clarify design intention at a high level. [MR97] In this sense, they may be a vehicle to progress, though they do

not promise to completely overcome the sufficiency problem for design specifications.

Another direction in architecture research attempts to relate architectures (in the solution space), to the application (problem) domain through *domain specific software architectures* (DSSA). [Tra95] However, application domains have not yet been classified and thus the relationships between architectures and application domains have not been established and cannot be evaluated.

c. Formal Specifications

Formal specifications directly relate to the problem at hand. Meyer claims that they "help expose ambiguities and contradictions because they force the specifier to describe features of the problem precisely and rigorously." [Mey85 at 22] Wing extols the ability to "detect discrepancies between a specification and an implementation." [Win90 at 20] But, though progress is being made [LHH94], it is recognized that formal specifications cannot yet adequately model nonfunctional properties such as safety. [Hal90] [Win90] Even assuming that progress is made, there are reasons that formal specifications will not be sufficient to identify arbitrary manufacturing defects in code.

As Jaffe observes, the power in a specification is in the abstraction: the ability to omit details in order to deal with complexity. [Jaf88] Thus, formal specifications only cover some of the program's behavior, they leave out details that are not considered important. The implementers must fill in the details where they've been

left out. [Hal90] Jaffe admonishes specifiers to leave little to the implementer's judgment when specifying safety-critical systems because implementers do not have the required systems background. [Jaf88] He is troubled by the reliance on human judgment in specification of the proper level of detail for the implementers.

Since the sufficiency of formal specifications relies on human judgment, it is subject to the same problems discussed in chapter four such as imperfect foresight and design realities only discoverable during implementation. [SB82] Thus, while the precision of formal specification may be an improvement in our ability to detect arbitrary manufacturing defects in the implementation, it is not a complete solution to the problem.

Though progress is being made, most software engineers understand this to be an unsolvable problem. [Lev95] It is *not a new problem*, the specifications for traditional physical artifacts experience many of the same difficulties. [Pet92]

A Difficulty Inherent to Software: Description as Product

The reality of software as a product is that it is actually the *description* of a machine, not the embodiment of the machine itself. This fact is troublesome when software engineers or lawyers need to distinguish software design intention from software construction in source code. It presents a difficulty not present in physical products such as automobiles or lawnmowers. For such physical systems, there is a natural boundary between design intention and construction imposed by the different properties of the media of expression for each stage, shown in figure 11:

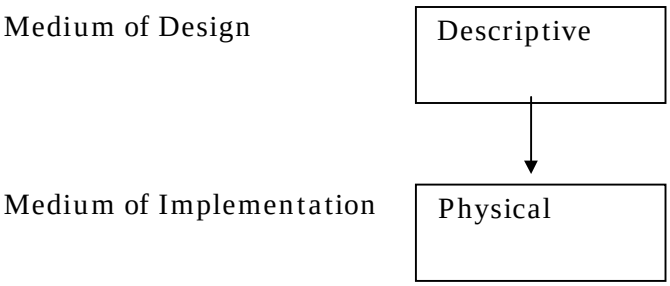

Figure 11 - Physical Systems

The ability of the auto assembler to perform any design or express individual intention in the product (make significant choices during implementation of the product) is severely limited, not only by the manufacturing process (strives for simplicity and uniformity in pursuit of efficiency) but by the very nature of the implementation medium: the physical "stuff" used in construction. The assemblers on the Chevy Cavalier line cannot possibly, one day, decide to increase trunk space for the car being constructed. They cannot add a new steering system, implement the brakes in a novel manner, cannot build a boat instead of a car. These choices would require major retooling, new plans, new processes; parts would need to be designed and built, then physically constructed. It is not easy to reshape the trunk lid to your liking! Once the metal is stamped or shaped, it is not easy to change. Changes in physical implementation usually take a major effort, for they are constrained by size, shape, weight and other factors that restrict design choices.

Now, notice how software construction is organized. The software specifications are descriptions of solutions to the problem under consideration. They may be written in structured english or some more formal medium. The implementations are also descriptions of solutions, but to the problems described by specifications. They are

written in a programming language: a human readable language that is capable of mechanical translation into machine readable form. As Jackson says, "our technology is the technology of description." [Jac96 at 17] The relationship between specifications and implementations for the software product is shown below in figure 12:

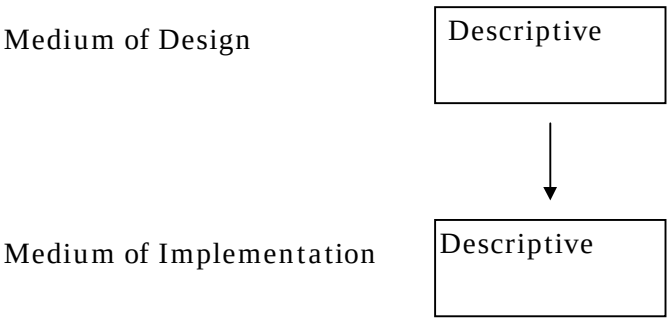

Figure 12 - Software

This observation has two important consequences for the construction of the software product:

- it *enables* the coder to do major design work; and,
- it *requires* the coder to be skilled in manipulation of a broad spectrum design medium.

Enabled for Major Design

For those constructing software, there are few constraints on implementation that are not present for software design. This non-physical medium of logical description

is not constrained by size, weight or resistance to major change. The coder is, within the scope of his programming assignment, as free as the designer to create the product implementation of his choice, virtually unrestricted by immutable physical realities as is the auto assembler. The coder is able to exert control over his implementation of the software product that the auto assembler cannot even dream of! Prime examples of the coder's ability to perform major design can be seen in the so-called "easter eggs" in mass produced software products. For instance, Microsoft Excel 97 contains an entire flight simulator accessible by several obscure keystrokes.[26]

Coding Skills Required

As has been seen, coding is not an entirely clerical skill. General unskilled labor is not considered sufficient to the task, as might be the case for the automobile assembly line. The coder must be skilled in working with the broad design medium of programming languages at the very least. Most coders that have a 4 year degree in computer science would be equally trained in code design issues.

Overall, the coders have the apparent ability to do major design, they are skilled in manipulation of the design medium, and, as shown previously in chapter four, are often forced to deal with design issues on a regular basis during code construction. It is no wonder that an objective separation of the specification from the implementation is recognized to be difficult!

[26] *Risks Forum*, 5 January, 1998. See www.CSL.sri.com/risksinfo.html for archive info. Also see http://www.eeggs.com/ for an extensive archive of examples.

Partial solutions are suggested by the research. The closer software engineers can get to a halting algorithmic solution to the problem of distinguishing specifications from implementations, the better for society when the accident cases come to court. However, there are even more fundamental problems with the technical approach to this problem.

More Fundamental Difficulties With Specifications

Notice that software specifications vary in level of detail on a continuum broadly outlined in figure 13. For purposes of interpretation, hold the code fixed. Now consider the variation in level of detail possible in specifications for that code.

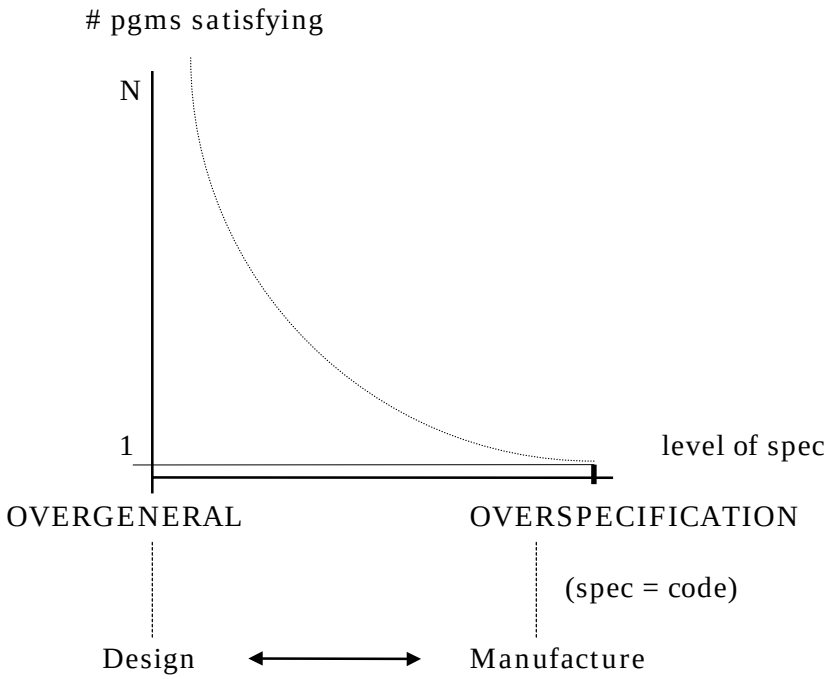

Figure 13 - Specification Detail

Though the magnitude on the graph may not be well defined, the general idea is clear: more trivial specifications allow a greater number of distinct programs to satisfy them, while more complete (more detailed) specifications will allow fewer distinct programs to satisfy them. The limiting case of the complete specification is that of specification = code, where there is only one program that satisfies the specification. Also note the additional dimension along the bottom of the graph, where the effect of these specifications on the identification of defects is noted.

If the design specification is made more trivial so that many distinct programs can satisfy it, then there is an argument based on the law that any defects would be

defects of design, including inadvertent mistakes in executing true design intention (implicit). Of course, one might also argue that such specifications are insufficient to capture much of the design intention, a design argument to begin with. The actual coding mistakes are swallowed by designs that tend toward triviality.

If the design specification is closer to the code and specifies more, then more deviations from true intention will fail to satisfy the specification. They fit the definition of defects in manufacture.

However, *a given piece of code does not have a fixed relationship to its specifications*, they can be written (or rewritten) with more or less detail. The legal implications of testing to specifications are therefore dictated by the level of detail employed in the specification. This is a subjective choice!

Uniformity of application of the law is not supported by such a regime, for different organizations may use different techniques and languages for specification, with differing implications for their ability to "find" defects in manufacture. This sets up various incentives to write specifications towards one end or the other on the continuum. In some situations, software specifications may be intentionally manipulated to produce desired legal results favorable to one party or the other. Therefore, the law cannot accept them as the only input to the process.

Society needs to find a way to operationalize its goals for the safety of products in a rational manner. However, the optimum level of detail is not known, nor whether there is an optimum level.

Summary of Important Points

In this chapter, software has been shown to be unique among products, and that uniqueness has implications for the legal responsibility of software designers. Progress in the field of software engineering is unlikely to perfect our ability to distinguish the two classes of defect because it is the essential nature of the artifact that stands in the way. Further, it is shown that technical progress, even if completely successful, does not address the requirements of the law.

Chapter Seven:

Conclusions and Future Work

Summary and Conclusions

This work has addressed the problem of identification of arbitrary violations of design intention (defects in manufacture) for the software product. Earlier chapters have shown:

- accountability for software defects will be established under the law of products liability in tort; and,
- the law of products liability in tort operationalizes a form of social risk management by its classification of product defects into those of:

    1. design intention (design); and,
    2. unintentional defects in implementation of design intention (manufacture).

The basis for this work is a hypothesis that a simple "stage of production" analogy could be used to classify software defects in a rational manner. Under this analogy, software specifications would evidence the design intention for the code. An

arbitrary code flaw could then be compared to the specifications. If specifications were violated, a manufacturing defect has been identified. If specifications were satisfied, a potential defect in design has been identified.

Investigation of the software development process reveals the following:

- there exists the potential for both classes of defect in software code; and,
- software specifications are insufficient to distinguish classes of defects in arbitrary cases.

Specifications, in general, are not sufficient for products of other technologies either. The law has developed a test to reveal design intention that is independent of specifications: the "deviation from the norm" test. However, due to the nature of software production, this test fails in the normal case. This failure points to the novel nature of manufacturing defects in code: they are *generic* to the product.

The independent test fails completely to identify any generic manufacturing defects in software. A test of comparison to software specifications can be used to identify many of these defects, depending on its degree of sufficiency. The sufficiency of software specifications is a subject of active research in software engineering. Several areas of active research contribute to specification sufficiency, but none provide it in full.

Therefore, the hypothesis is disproved, the stage of production analogy fails to enable a rational classification of arbitrary software defects. It fails because software engineers cannot deliver full sufficiency of specifications for arbitrary fragments of code. This is true for all artifacts of design, though. The reason that it presents such a problem for the software product is that the "deviation from the norm" test fails to identify the generic defects in software manufacture.

However, it appears that progress in software specifications could take us as close as we like to full sufficiency, limited by time and resources. Perhaps they could be made sufficient "enough" to distinguish defects in most all practical cases.

Investigation of this possibility reveals deeper problems with the notion of software specifications as an objective repository of design intention. For traditional products like automobiles, the medium of implementation is defined by physical laws that constrain the implementers in their ability to deviate from the designs specified to something within well understood limits. Software implementers are not so constrained. In fact, the medium of software implementation is so similar to the medium of software design that implementers are quite as able as the designers to produce major deviations from the basic design intention (or to create their own).

Since this is so, the distinctions between the code implementation and its specification are those set by the software engineers. With current understandings of software as a product, the distinctions are not objectively imposed by nature.

The law cannot rely on subjective technical distinctions to reach the rational, repeatable results required under the common law.

Contributions of the Work

This work makes contributions to the areas of software engineering as well as products liability in tort. Contributions in the area of software engineering include:

1. Products liability issues are virtually certain to arise for software products. They may have enormous impacts for those involved in a case and eventually for the entire industry. This work presents the first detailed application of the legal defect classifications to the software product.
2. Risk management has emerged as an important component of the software development process. Current work in this area has not addressed expected costs of liability for product defects. The basic elements needed to address this issue are presented in this work.
3. Software engineers generally accept the proposition that specifications do not contain full design intention for the implementation. The limitations are often seen to be human fallibility and poor resource allocation, against which progress may be made by new tools and methods. This work presents a model of software that exhibits theoretical, as opposed to practical, limitations on progress in specification sufficiency. Progress in the use of specifications to guide

implementation must be tempered by the knowledge that distinctions between specifications and code are subjective in nature.

In addition to the above contributions, contributions to legal scholarship include:

1. The law of products liability is virtually certain to face the classification of software defects problem in the near future. Analyses in the legal literature have not dealt with the technical aspects of this issue to any depth. This dissertation presents a legal analysis based on a broad, detailed technical perspective. Such an analysis is of great value for the common law because the Courts are limited in the general breadth of possible analyses for each individual case. For example, Courts are reactive to problems, they only handle cases that are sued, they cannot proactively attack an issue as this dissertation does. Courts can only decide the specific issues presented for decision or review. This work takes a broad view of the larger issues surrounding software defects. Courts respond to the subjective arguments of the parties in a case. This work can take an objective view of the basic issues.
2. This dissertation articulates a new class of product defect, the *generic manufacturing defect*, for the software product. It is this class of defects shown to defy both legal and technical means at classification. It is thus to this class that legal attention must turn in order to apply social goals of risk management in some rational manner.

Future Work

Many lines of research are suggested by the current work:

1. The software process does not currently take legal requirements into account explicitly. This work suggests that requirements due to products liability in tort can be viewed as the requirements of "remote customers" of safety-critical consumer software. This view allows a direct and natural integration of these important considerations in the software process.
2. A related way to show software engineers that legal requirements are to be considered explicitly during the development process is to analogize the safety-critical software process to a process of experimentation. As experimenters, software engineers are obligated to obtain the consent of the subjects: the general public. The notion of legal consent is naturally contained in the requirements of the law of products liability in tort and can be directly related to case outcomes this way.
3. This work could naturally be extended to catalog software defect detection methods by the class of potential defects that they uncover. This could be beneficial for software risk management: expected costs of the defects could be estimated and the cost of the defect detection by a given method compared. The methods could then be applied in a more cost effective manner during the software process.
4. Former work engaged the idea of "process fault trees" [TRK] that dealt exclusively with the idea of design defects and negligence during the software

process. This work could be extended to deal with both kinds of defects and add relative expected costs in order to produce priorities along the branches that correspond to the most cost effective paths.

Bibliography

[Ban94] Banks v. ICI Americas, Inc, 450 S.E.2d 671 (Ga. 1994)

[Bei96] Beizer, Software IS Different, *Address at Quality Week '96 Conference* (1996)

[BD81] Brannigan, Dayhoff, Liability for Personal Injures Caused by Defective Medical Equipment, 7 *American Journal of Law and Medicine* 123 (1981)

[Bir80] Birnbaum, Unmasking the Test for Design Defect: From Negligence [to Warranty] to Strict Liability to Negligence, 33 *Vanderbilt Law Review,* 593 (1980)

[Boe88] Boehm, A Spiral Model of Software Development and Enhancement, *IEEE Computer,* May, 1988

[Boe90] Boehm, *Software Engineering Economics,* Prentice-Hall, 1981

[Bro95] Brooks, *The Mythical Man-Month, Anniversary Edition,* Addison-Wesley, 1995

[Con87] Conley, Tort Theories of Recovery Against Vendors of Defective Software, 13 *Rutgers Computer and Technology Law Journal* 1 (1987)

[Cro85] Cronin, Consumer Remedies for Defective Computer Software, *28 Journal of Urban and Contemporary Law*, 273 (1985)

[Dav95] Davis, *201 Principles of Software Development*, McGraw-Hill, 1995

[Eis88] Eisenberg, *The Nature of the Common Law*, Harvard Books, 1988

[Flu85] Fluor Corporation v. Jeppsen and Co., 170 Cal. App. 3d, 468 (Cal. App. 1985)

[Gem81] Gemignani, Products Liability and Software, 8 Rutgers Computer and Technology Law Journal, 173, (1981)

[GG75] Goodenough, Gerhart, Toward a Theory of Test Data Selection, *IEEE Transactions on Software Engineering*, June 1975

[GL81] Gemignani, *Law and the Computer*, CBI Publishing Company,

Inc. MA, 1981

[GS93] Garlan, Shaw, *An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering,* volume I. World Scientific Publishing, 1993

[Ham92] Hamlet, Are We Testing for True Reliability? *IEEE Software,* July 1992

[Hen76] Henderson, Design Defect Litigation Revisited, 61 *Cornell Law Review,* 541 (1976)

[Hen60] Henningsen v. Bloomfield Motors, Inc., 161 A. 2d 69 (NJ 1960)

[HT91] Henderson, Twerski, Closing the American Products Liability Frontier: The Rejection of Liability Without Defect, 66 *NY Law Review,* 1263 (1991)

[IEE94] *IEEE Standard Glossary of Software Engineering Terminology,* Institute of Electrical and Electronics Engineers, Inc. 1994

[Jac95] Jackson, *Software Requirements and Specifications,* Addison-Wesley, 1995

[Jaf88] Jaffe, *Completeness, Robustness, and Safety in Real-Time Software Requirements Specifications: A Logical Positivist Looks at Requirements Engineering,* Ph.D. Thesis, University of California, Irvine, 1988

[Kan95] Kaner, Software Negligence and Testing Coverage, *The Software QA Quarterly,* Vol. 2, No. 2, 1995

[Lev95] Leveson, *Safeware,* Addison Wesley, 1995

[LHH94] Leveson, et.al., Requirements Specification for Process-Control Systems, *IEEE Transactions on Software Engineering,* Vol. 20, No. 9, September 1994

[LT93] Leveson, Turner, An Investigation of the Therac-25 Accidents, *IEEE Computer,* Vol 26, No. 7, July, 1993

[Mey85] Meyers, On Formalism in Specifications, *IEEE Software,* January 1985

[Miy92] Miyaki, Comment: Computer Software Defects: Should

Computer Software Manufacturers Be Held Strictly Liable for Computer Software Defects? *8 Computer and High Technology Law Journal*, 121 (1992)

[Mor89] Mortimer, Note: Computer-Aided Medicine: Present and Future Issues of Liability, 9 *Computer Law Journal*, 177 (1989)

[MR97] Medvedovic, Rosenblum, Domains of Concern in Software Architectures and Architecture Description Languages, *Conference on Domain-Specific Languages, USENIX Association*, October 15-17, 1997

[NLJ91] Slind-Flor, Supplier Pulls Software - Ruling Causes Uproar *National Law Journal*, July 29, 1991, page 3

[NW82] Nelson and Winter, *An Evolutionary Theory of Economic Change*, Belknap Press of Harvard University Press, 1982

[Nyc79] Nycum, Liability for Malfunction of a Computer Program, 7 *Rutgers Journal of Computers, Technology and Law*, 1 (1979)

[Owe80] Owen, Rethinking the Policies of Strict Products Liability, 33 *Vanderbilt Law Review*, 681

[Owe96] Owen, Defectiveness Restated:" Exploding the "Strict Liability" Products Liability Myth, *1996 U. Ill. L. Rev.* 743

[Par86] Parnas, A Rational Design Process: How and Why to Fake It, *IEEE Transactions on Software Engineering,* Vol SE-12, No. 2, February 1986

[Par90] Parnas, Evaluation of Safety-Critical Software, *Communications of the ACM,* Volume 33, No. 6, June 1990

[Pet92] Petroski, *To Engineer is Human,* Vintage Press, NY, 1992

[Pin81] Grimshaw V. Ford Motor Co., 174 Cal. Rptr. 348 (Cal. Ct. App. 1981)

[Pou49] Pound, *The Spirit of the Common Law,* Beacon Press, 1949

[Pre84] Prentice v. Yale Mfg. Co., 421 Mich. 670 (Sup. Ct. MI, 1984)

[Pro84] Prosser, Keeton, *Prosser and Keeton on Torts, 5th Edition,* West Publishing, MN, 1984

[RAO92] Richardson, Aha, O'Malley, Specification-based Test Oracles for Reactive Systems, *Proceedings of the ACM SIGSOFT '89, Third Symposium on Software Testing, Analysis, and Verification (TAV3),* Key West, FL, Dec. 1989

[Res98] *Restatement Third, Torts: Products Liability,* American Law Institute Publishers, MN, 1998

[Rix86] Rix v. General Motors Corp, 723 P.2d 195 (Mont. 1986)

[RM98] Robbins, et. al, Integrating Architecture Description Languages with a Standard Design Method, *Proceedings of the Twentieth International Conference on Software Engineering* (ICSE 98, Kyoto, Japan), IEEE Computer Society Press, April 19-25, 1998, pp. 209-218

[Roy70] Royce, Managing the Development of Large Software Systems: Concepts and Techniques, *ICSE 9 Proceedings* (1970)

[RT88] Richardson, Thompson, The Relay Model of Error Detection and its Application, *IEEE Transactions on Software Engineering,* June 1993

[SB82] Swartout, Balzer, On the Inevitable Intertwining of Specification and Implementation, *Communications of the ACM*, Vol. 25, No. 7, July 1982

[Sch96] Schmidt, The Impact of Social Forces on Software Project Failures, editorial, *C++ Report*, April 1996

[Sch99] Schach, *Classical and Object-Oriented Software Engineering*, McGraw-Hill, 1999

[Smi94] Smith v. Keller Ladder Co., 645 A.2d 1269 (N.J. Super. Ct. 1994)

[Som92] Sommerville, *Software Engineering,* Addison-Wesley, 1992

[Tra65] Traynor, The Ways and Meanings of Defective Products and Strict Liability, 32 *Tenn. L. Rev.* 363 (1965)

[Tra97] Tracz, DSSA (Domain-Specific Software Architecture) Pedagogical Example. *ACM Sigsoft Software Engineering Notes*, July 1995.

[TRK96] Turner, Richardson, King, Legal Sufficiency of Testing Processes,*Proceedings of the 15th International Conference on*

Computer Safety,Reliability, and Security, Vienna, Austria, Oct. 1996

[TWDP76] Twerski, et. al., The Use and Abuse of Warnings in Products Liability, Design Defects Litigation Comes of Age, 61 *Cornell Law Review* 495 (1976)

[Voa97] Voas, A Crystal Ball for Software Liability, *IEEE Computer,* June 1997

[Web92] Weber, Bad Bytes: The Application of Strict Products Liability to Computer Software, 66 *St. John's Law Review,* 469 (1992)

[Win91] Winter v. G.P. Putnam's Sons, 938 F.2d 1033 (9th Cir. 1991)

[Wit85] Witherell, *How to Avoid Products Liability Lawsuits and Damages,* Noyes Publications, 1985

[Wol93] Wolpert, Product Liability and Software Implicated in Personal Injury, *Defense Counsel Journal,* 519, October 1993