The Increasing Threat of Legal Liability for Software Developers

JANICE C. SIPIOR
Villanova University, USA

BURKE T. WARD,
Villanova University, USA

WILLIAM P. WAGNER
Villanova University, USA

This paper discusses the increasing threat of legal liability for developers should software malfunction and cause financial loss or harm to the user. Recent events in the software industry have signaled a changing environment for development organizations. A discussion of the mounting market expectations for software to function properly is presented to underscore the increasing potential for users to seek legal recourse. Various theories in the U.S. legal system may form the basis for legal action on the part of the user, based upon the characteristics of the individual case. The focus of this discussion is on software which is developed for sale, as opposed to in-house development, since most legal actions would arise between a vendor and purchaser. Although we conclude that the legal liability of the software developer is currently unclear and varies by jurisdiction, the threat is nonetheless present and may indeed be on the rise.

Software contributes to improvements in organizational efficiency, supports managerial decision making, and aids in gaining or maintaining competitive advantage. The resulting demand for software has attracted development organizations to the lucrative commercial and retail market. Software may be mass marketed canned software sold at retail under a shrink-wrap, included as a part of turnkey systems which require either minimal or no in-house modifications, or specially developed for systems designed to fulfill user’s particular needs. Tailored systems may be developed by an internal systems development team or may be outsourced, but the trend toward outsourcing is rising (Lacity and Hirschheim, 1993). Developing software for use outside of the development organization may open the floodgates to potential legal liability. If software is not deemed to function properly, do the users have any legal recourse? Are programmers, system analysts, Information Systems (IS) managers, or organizations involved in development efforts exposed to legal liability? Many developers are rightly concerned about the extent of their liability should the software they developed and sold malfunction (Samuelson, 1993).

This paper seeks to raise the awareness of programmers, system analysts, IS managers, and organizations involved in software development about the increasing threat of legal liability should software malfunction and cause financial loss or harm to the user. A discussion of the mounting market expectations for software to function properly is presented to underscore the increasing potential for users to seek legal recourse. Various theories in the U.S. legal system may form the basis for legal action on the part of the user, based upon the characteristics of the individual case. These are reviewed to inform developers of the potential bases for cases against
them. The focus of this discussion is on software which is developed for sale, as opposed to in-house development, since most legal actions would arise between a vendor and purchaser. Although we conclude that the legal liability of the software developer is currently unclear and varies by jurisdiction, the threat is nonetheless present and may indeed be on the rise.

**Expectations of Properly Functioning Software**

Should software be expected to contain errors or function properly all of the time? Flaws, resulting in both trivial and dire consequences, have plagued software for decades. A classic example of a serious malfunction occurred in a U.S. nuclear missile warning system on October 5, 1960 (Belsie, 1994). Radar sensor input from Thule, Greenland was erroneously interpreted as a massive attack by the then Soviet Union, with a certainty of 99.9 percent. The actual cause of the warning system to issue the attack alert was identified, precluding nuclear warfare. The rising moon had caused echoes from the radar sensors, a factor overlooked during development. Unfortunately, software defects are not always detected in advance of disaster. Perhaps one of the most tragic incidents associated with software malfunction occurred in a system developed by Atomic Energy of Canada Ltd. to control radiation doses delivered to cancer patients (Byte, 1995). On-screen editing performed with the up-arrow key caused two modes of operation to be mixed, resulting in a radiation dose more than 100 times higher than the average dose (Joyce, 1987). At least four patients believed to have received the erroneous radiation overdose died; others were seriously injured.

Software errors have become more prevalent, ranging from financial loss to life-threatening situations (Henderson, 1995). The reason for the increasing number of software defects is not just because software itself is more prevalent (Gianturco, 1995). A widespread move away from highly standardized mini or mainframe systems to networks, comprised of various brands of hardware and software connected across long distances, has occurred. Coupled with the complexity of such networks is more flexibility in user interaction, marked by the emergence of Microsoft’s Windows. Various tasks can be performed with no predetermined sequence or combination of events, unlike old programs which performed tasks through a structured series of command sequences. The new style software requires exhaustive testing to assess every possible sequence, permutation, and combination of events, which is virtually impossible (Diefenbacher, 1995). In response, the automated testing market has experienced rapid growth, projected to double sales in one year to $100 million (LaMonica, 1995). Automated testing tools capture the input of a human tester and generate test scripts to be run repeatedly. Errors are detected and testing resumes once the cause is determined and the fault repaired. However, each subsequent error is more difficult to detect and correct. Although automated testing tools are increasingly available, only about 75% of the code in the 60 leading products in the software industry has been tested (Henderson, 1995). In the overall development community, only about 35% of the code in a typical application is tested (Henderson, 1995). The top four development organizations, however, have been reported to be committed to quality development, detecting up to 95% of software defects before delivery to users (Henderson, 1995).

In the highly competitive software market, development organizations are driven to develop software rich in both features and quality, within a tight schedule (Moskun, 1996). Even developers with the strictest quality control may distribute software with some remaining defects. What defects remain may or may not be known by the developer before the software is released. Users have reported frustrating instances of hours wasted in attempting to get software to perform some simple task, only to have the development organization finally admit the software will not function in the manner desired due to a known problem (Foster, 1996a). Negative effects on sales may force the industry to inform users of known defects (Rigdon, 1995a).

A legal implication of the argument that some level of software defects may be inevitable is the proposal for revisions to the Uniform Commercial Code (UCC). Disclosure of known defects could ultimately be required by law, should proponents be successful in including this provision in the proposed Article 2B of the UCC (Foster, 1996b). Development organizations would be held liable should they not make a good faith effort to appropriately disclose known defects. This approach may be preferable to both developers and users. If known defects are disclosed appropriately, the developer’s liability could be reduced since the user could read about and plan for any potential defects or incompatibilities in advance of software purchase or use. To disclose known defects may seem like a simple request to developers, but what exactly is a defect, as opposed to a design decision, a feature, or a performance limitation? Since no standard exists in the software industry for certifying the quality of software, the determination remains a matter of personal opinion (Patterson, 1990). The increasingly complex computing environment has made it more difficult to develop complex systems, correct in terms of the design specifications, with no defects. Even if it were possible, the development team would certainly be unable to foresee and accommodate all unanticipated circumstances which may arise during use. “Are we justified in saying that the designer, who operated without the benefit of hindsight, can be held liable for the flaw?” (Denning, 1990). Many software errors have been attributed to human error, rather than to the design (Norman, 1990). “Software’s written by people and people make mistakes” (Joyce, 1987). Will users accept the inevitability of flawed software without seeking recourse, even in cases of devastating consequences?
Clearly, as software and the environment within which it operates have become more complex, the potential for errors has increased. At the same time, pressure for software to function properly has been building within the computer software industry (Mossberg, 1995). A number of decades ago, the term “bug” was first applied to software defects. This cute little term subsequently became widely accepted, carrying with it both expectations for its occurrence and acceptability for its presence. However, the tide is turning.

Examples Signaling Change

The leaders in the software industry, Microsoft Corp., Novell Inc., and IBM’s Lotus Development Corp., are no longer the charming fledgling start-up companies they once were. The industry leader, Microsoft Corp., for example, is no longer based in an Albuquerque apartment, but is headquartered in 26 buildings in a spacious park-like setting in Redmond, WA. The company’s chairman, Bill Gates, astutely built Microsoft into the multi-billion dollar company it is today (Button, 1995). With over $6 billion in sales, Microsoft employs 17,000 worldwide (Angrist, 1995), including a battalion of engineers, marketers, attorneys, and public relations personnel, to promote and deliver quality products to their customers, just like any other product manufacturer.

With such industry changes, has come a difference in what the software market is willing to accept. Gone are the days when users were so delighted the computer was able to do anything at all that they were willing to accept the inevitability of program bugs. A recent occurrence in the computer hardware industry has signaled changing market expectations. Intel Corp., the leading chip manufacturer, knowingly produced Pentium chips which were defective. Intel first denied there was an error. Then, admitting there was an error, Intel refuted the need to offer replacement chips since functionality was adversely affected only in rare instances, involving floating-point operations. Customers would have to prove they could be affected by such miscalculations and Intel would decide on a case-by-case basis whether to provide a replacement chip. Contrary to Intel’s claim that mistakes were limited to complex mathematical computations only encountered by scientists, “errors might conceivably affect business people designing currency trades, insurance contracts, and even bridges” (Fortune, 1994). Intel had known about the flaw months before a mathematics professor discovered it in October 1994, but believed errors resulting from it were unlikely (Hill, 1995). Following market outrage, Intel was forced to replace, free of charge, all defective Pentium chips, at an estimated cost of $475 million (Hill, 1995). This pivotal event has had a spill-over effect for software producers. Indeed, the sands have shifted under the industry’s feet.

One of the more well-known examples within the software industry concerns Intuit’s TurboTax, the company’s leading tax assistance software. When problems in software use were revealed a few weeks before the April 17, 1995 individual tax filing deadline (April 15 was a Saturday), Intuit had initially presented the same argument as Intel. The defect would affect functionality adversely only in rare instances, involving particular financial considerations. Again, market forces led Intuit to revise and replace TurboTax software free of charge. Further, Intuit promised to pay any Internal Revenue Service penalty, plus interest, resulting from errors attributable to TurboTax use. Intuit admitted that in any given year, about 1 percent of its users are affected by flaws in its tax preparation software (Lewis, 1995). The resulting cost to Intuit for the penalty guarantee has been reported as less than $1,500 over three years (Rigdon, 1995b). Intuit simply had no time for extensive testing during systems development. The development schedule was dependent upon Congress to finish its annual tax code revisions and the release of final versions of Federal and state tax forms by late December. User demand for tax preparation assistance has to be met by about mid-January to allow sufficient time for use before the personal tax filing deadline.

An example demonstrating just how far-reaching software problems can be, impacted mainstream America on Christmas day in 1994 and was even characterized as “the first bug to affect popular culture” (Byte, 1995). The much anticipated CD-ROM version of the popular “The Lion King,” a children’s multimedia entertainment package, was marketed for home use by Disney Interactive, the software unit of Walt Disney Co. With sales topping 200,000, Disney was deluged with calls from disappointed parents who unsuccessfully tried to install the software on Christmas day. Days and weeks went by before Disney responded to the calls, mail, e-mail, and bulletin board messages. When it finally did respond, the initial stance was to claim that problems were caused by novice users who did not know how to install the software properly because they did not read the message on the box and their equipment was obsolete. Within a month, Disney changed its tone and increased its product support staff from eight to about 50, at an unanticipated expense. New versions of the CD-ROM were sent to customers who did not have the required sound card. Disney later conceded that it released the CD-ROM even though it knew errors were present, believing only a small percentage would be affected (Rose and Turner, 1995). Outside developers, Media Station and Microsoft Corp., who had produced portions of the code, informed Disney of the program’s defects. Trying to capitalize on market demand, resulting from re-release of the movie upon which the CD-ROM was based and the Christmas season, Disney decided to release it anyway.

The examples presented are indicative of a shift which is occurring within the software industry. “Now we are entering an era in which many buyers are uninformed consumers,” (Diefenbacher, 1995). These consumers are not sophisticated computer users. Programming is completely foreign to them and the idea that software could produce erroneous results is totally unexpected. Although less publicized than the mass
marketed shrink-wrapped software malfunctions are a multitude of examples which entail turnkey systems and specially-developed systems designed to fulfill a user’s particular needs. Malfunctions with these systems certainly affect fewer users, but the impact of market outrage is still the same for software developers.

A Shift from Caveat Emptor toward Caveat Venditor

Is the argument presented by Intel, Intuit, and Disney acceptable to users of software? An analogous argument for an automobile manufacturer would be the contention that a known brake failure only occurs if a driver were to suddenly swerve right, then left, at a particular speed, on a rainy day, with a specific brand of tires. Such an occurrence is certainly rare, but is nonetheless a product defect. Users, like any other consumers, are beginning to treat defects within the software industry as they do defects in other purchases. If it doesn’t work properly, take it back. If it causes financial loss or physical harm, seek monetary recovery within our legal system. Software is not usually thought of as being associated with physical harm, but the risk of such injury from defective systems is increasing as software enters into more aspects of our lives. The areas of medical technology, transportation systems, industrial plants, nuclear power stations, and telephone networks are fraught with a heightened possibility of physical injury from improperly functioning software (Peterson, 1991).

Increases in the demand for a diversity of applications by both commercial and retail markets have changed the legal environment of development organizations from the comfortable world of caveat emptor toward the high accountability of caveat venditor. A move along the Marketing Ethics Continuum, represented in Figure 1, is occurring. This continuum provides benchmarks against which software developers may gauge their responsibility to users as the industry matures. A new industry operates at the caveat emptor position which represents the view of “buyer beware,” since profit maximization is the initial basis for evaluating successful business endeavors (Smith, 1995). For users of software, this means it is the responsibility of users to find the resolution to problems and difficulties encountered in software use. Just over a decade ago, change within the software industry was recognized by Edward M. Esber, Jr., then President of Ashton-Tate Corp., when he stated “In the early days of the business, the purchasers were technically oriented people...more willing to accept a fault in a product. Today it’s more of a consumer market” (Davies, 1985). Momentum for change continued as consumers were urged five years later “to say ‘no’ to companies that deliver minimally tested, bug-riddled products,” (Patterson, 1990). Today, caveat emptor is no longer an acceptable stance for software developers. Although developers’ motives still “often have more to do with marketing than with quality assurance” (Vadlamudi, 1995), a shift toward the caveat venditor end of the continuum is evident through increased scrutiny and higher standards on the part of users evaluating the performance of purchased software.

As market expectations force the software industry away from caveat emptor, software developers correspondingly must become more user oriented. The industry practice position represents the initiation of business actions dealing with improving user satisfaction. These business actions become norms and develop into industry practice. For example, technical support desks for users have become common in the software industry, although many organizations charge for this service (Rigdon, 1995a). As the development of industry practice advances, ethics codes, the next position on the continuum, form. Various entities, such as individual companies, industry groups, and professional organizations, formalize industry practice for improving user satisfaction as ethics codes. For example, the Association for Computing Machinery (ACM) has adopted a Code of Ethics and Professional Conduct for its members to “serve as a basis for ethical decision making in the conduct of professional work” (ACM Executive Council, 1992). For software developers, this Code advocates quality in both the process and products. An even greater focus on user satisfaction is represented by the consumer sovereignty position. At this point on the continuum, a software developer promotes the user’s interests in business transactions by ensuring that users have the capability to make informed purchase choices. For example, information concerning known defects could voluntarily be made available to potential customers.

At the final endpoint of the continuum is caveat venditor, which represents “seller beware.” At this position, software developers strive to maximize user satisfaction. An emphasis on the user and user satisfaction has been recognized as an important way to reduce the developer’s exposure to legal liability (Mykytyn et al., 1995). Software developers who are not ethically responsible in business dealings with users may be held accountable through the U.S. legal system (Diefenbacher, 1995). Developers must thus be prepared for the changing expectations of users, emphasizing the need for

<table>
<thead>
<tr>
<th>Caveat Emptor</th>
<th>Industry Practice</th>
<th>Ethics Code</th>
<th>Consumer Sovereignty</th>
<th>Caveat Venditor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer interests favored</td>
<td>Consumer interests less favored</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
developers to understand the legal environment to which they are subject.

**Legal Theories Applicable to Software Malfunctions**

The development of software for sale constitutes a business transaction, subject to various elements within the U.S. legal system. This section examines the legal theories applicable should software sold for use outside of the developer’s organization malfunction. The user may seek legal recourse under one or more of the theories within our legal system, depending upon the circumstances of the case. The following analysis, summarized in Table 1, addresses breach of contract and the tort theories of fraud, negligence, and strict liability in tort.

**Breach of Contract**

Software sold for use outside of the developer’s organization is often governed by a contract prepared by the developer, to serve as an agreement between the developer and the user. Issues which should be specified in the contract include responsibility for maintenance, system performance, and service levels (Lacity and Hirschheim, 1993). Should the stipulations of the contractual agreement not be upheld by the developer, that is the developer breaches the contract, the user may seek a legal remedy.

A fundamental question in any legal action based on breach of contract is whether the subject matter of the contract, i.e., the software, is within the scope of Article 2 of the UCC or subject to common law principles. If the contract is within the realm of Article 2, either the user or the software developer may find this result preferable, depending on the circumstances of the case. The user may have greater substantive rights, than he would otherwise have under common law, through the implied warranties of merchantability (UCC Sec. 2-314) and fitness for a particular purpose (UCC Sec. 2-315). Both of these implied warranties could provide benefits to the user which are beyond those stated in the contract. Conversely, the developer may benefit by being permitted to limit his liability or limit the remedies available to the user. The developer could also disclaim implied warranties, that is, exclude the warranties provided through our legal system under Article 2, thereby limiting his liability for defective software only to the terms expressly stated in the contract.

**Article 2 of the Uniform Commercial Code (UCC)**

Article 2 standardizes transactions involving goods between states which have adopted it. A threshold question in determining the applicability of Article 2 to software cases is whether the contract is for goods or for services. For Article 2 to apply, the contract must relate to a transaction in goods (UCC Sec. 2-102). Since software usually involves both physical product and services, the courts are not in agreement concerning the applicability of Article 2. Computer hardware is clearly considered a good, and thus within the Article’s scope. The trend for software transactions has been to find that the sale of goods aspect predominates and the service aspect is more incidental to the contract (Advent Systems Ltd. v. Unisys Corp., 1991). Although software is a codified form of intangibles such as data, information, or knowledge (service), it has traditionally been sold on a physical disk, and thus could arguably be a good within Article 2. An analogy supportive of this argument is to a professor’s lectures (service) being published as a book (goods), which was used in a case dealing with a cookbook (Cardozo v. True, 1977). It should be noted that because Article 2 was not designed to address information technology, the National Conference of Commissioners on Uniform State Laws is currently drafting a proposal for a new Article 2B.

Assuming a software contract is within Article 2, an initial question is what warranties may be a part of that contract. Article 2 defines contractual warranties as either express or implied. Either of these may provide the basis for a breach of contract on the part of the developer. The remedies a user is able to obtain may be restricted by limitations and disclaimers the developer has included in the contract.

**Express Warranties**

Under Article 2, an express warranty is an affirmation of fact or promise relating to the goods, any description of the goods, brochures, samples, or models that becomes a part of the basis of the agreement between a developer and user (UCC Sec. 2-313). The use of words such as warranty or guarantee by the developer are not required to form an express warranty. For example, a demonstration showing a user’s equipment working with a particular type of computer was held to be an express warranty as to communication capabilities (Cricket Ally v. Data Terminal, 1987). In this case, the developer did not include an integration clause in the written contract. Such a clause would state that the entire agreement is contained within the written contract only, thus excluding the capabilities shown in the demonstration. The inclusion of an integration clause may be problematic for the user, since any prior demonstrations, representations, or statements could not be interpreted as express warranties. For example, if a developer orally states that his software will require minimal training time for employees, but does not include that statement in the written contract, an express warranty may not apply. The effect of the integration clause, in this example, is to disclaim prior representations and warranties not contained in the written contract.

**Implied Warranties**

Implied warranties are created by operation of law and not by the express representations of the developer. For software, the applicable warranties include the implied war-
ranty of merchantability (UCC Sec. 2-314) and the implied warranty of fitness for a particular purpose (UCC Sec. 2-315). The implied warranty of merchantability essentially indicates that the product must be of fair or average quality. For software, how would such a quality level be assessed? Perhaps it might depend on the area of application. For life-critical applications for example, a higher degree of reliability is required since the stakes are literally a matter of life and death (Peterson, 1991).

The implied warranty of fitness for a particular purpose means the product must be fit for the user’s intended purpose. The developer must have known the intended purpose for which the software was needed, and that the buyer was relying on the developer’s skill and judgment. This warranty is more likely to apply to specific-purpose software that is designed for a particular use, regardless of whether it is custom designed for one particular user or developed to fulfill a specific need in the mass market. For example, if software developed for a particular user did not perform as it was custom designed in a set of program flowcharts, the user might have a claim for breach of this implied warranty. Similarly, if mass-marketed software designed for the specific purpose of individual income tax preparation failed to produce the income tax return as required by the Internal Revenue Service, the user may again have a claim. The implied warranty of fitness for a particular purpose could deem the respective software to be inappropriate for the specific task it is intended to support, resulting in a ruling more favorable to the user.

These two implied warranties may merge when applied to software. The user is usually relying on the skill and judgment of the developer, who may be aware of the user’s requirements and needs. Thus, the software must not just be of average quality, but meet the user’s specific purposes, of which the developer or vendor was aware (Hollingsworth v. The Software House, 1986).

**Disclaimers and Limitations of Remedies**

If the contract for software is breached by the developer,
Article 2 provides the user with a variety of contractual remedies. A common remedy is general compensatory damages, such as recovery of the purchase price. In addition to general compensatory damages, a user may be entitled to consequential damages for economic losses resulting from the user’s particular needs not having been met, of which the vendor knew or had reason to know at the time of the contract (UCC Sec. 2-715 (2) (a)). Consequential damages usually include lost profits, but may also include other losses and expenses, such as extra labor costs, allowing the user to be awarded a larger financial settlement. For example, in the previously mentioned Cricket Alley Corp., (1987) case, the system developer, Data Terminal, was ordered to pay $78,781.79 in economic damages resulting from Cricket Alley’s increased labor costs associated with trying to overcome flaws present in a cash register system.

This potential open-ended liability of the developer is usually restricted by protective provisions in the developer’s contract. For example, it is common in the software industry for the developer to disclaim the aforementioned implied warranties. Article 2 permits such disclaimers if certain technical requirements are met (UCC Sec. 2-316). Further, the remedies available to the user, including consequential damages, may be restricted by limitations contained in the developer’s contract (UCC Sec. 2-719). In the software industry, it is not uncommon to find the user’s remedies limited to replacement with an upgraded version, or to have the amount of damages limited to the purchase price, thus protecting the developer against substantial consequential damages. For example, if a $50,000 system is defective and causes the user’s business to fail, such a clause would limit the user’s recovery to $50,000, not the full amount of the user’s loss.

Article 2 affords some protection to the user from disclaimers and limitation of remedies/damages provisions placed in the contract by the developer. If these provisions are found to be either unconscionable or cause a limited remedy to fail of its essential purpose, then the restrictive clause may be ignored and all other remedies become available (UCC Sec. 2-302, 2-719). Although unconscionability is not defined in Article 2, it generally deals with the prevention of oppression and unfair surprise, and not changes in the allocations of risks because of superior bargaining power (UCC Sec. 2-302 Comment 1). For example, a clause in a contract which releases the developer from liability for physical harm suffered by the user might be judged to be unconscionable. For software contracts which are commercial rather than consumer, unconscionability is not typically found since the parties are usually business persons who have negotiated the terms of the contract (Consolidated Data Terminals v. Applied Digital Data Systems, 1983). Similarly, if a limitation clause will cause a limited remedy to fail of its essential purpose, i.e., provide a remedy which is not an effective remedy, the limitation clause is disregarded (UCC Sec. 2-719). For example, if in a software contract, the user’s remedy was limited to replacement with an upgraded version, and a debugged upgrade could not be developed, then the clause fails of its essential purpose and is not enforced.

Article 2 permits software developers to greatly restrict their potential liability for defective software. As a result, users incurring financial loss or physical harm from use of the software have sought to circumvent these contractual limitations by suing the developer under various tort causes of action. While most litigation addressing software defects is based on some form of breach of contract theory, tort based claims are on the rise.

Tort Theories

A tort is a private or civil wrong other than a breach of contract, for which a court may provide a remedy (Black, 1979). As previously discussed, most software contracts are prepared by the developers and contain possibly enforceable disclaimers, limitation of remedies, or limitation of damages clauses. Particularly troublesome to a user who allegedly suffers from financial loss or harm are contractual provisions that prohibit the recovery of consequential damages, e.g., lost profits. Tort-based legal action attempts to circumvent these disclaimers and limitations because the user’s recovery is based on a specific tort, and not on the contract. While there are numerous possible tort theories, fraud and negligence (computer malpractice) appear to be the most common. Tort recovery is commonly applied to users who have suffered physical harm, but is often not an option available to the user whose loss is economic, that is loss from business interruption and lost profits. A user suffering economic loss may only be able to seek a recovery based on the provisions expressly stated in the contract (Apollo Group, Inc. v. Avnet, Inc., 1995; Wolpert, 1993). For example, clauses in a developer’s contract limit the user’s remedy, should the software not work properly, to an upgraded version or the return of the purchase price, whichever the developer chooses. The user, however, incurred lost profits of $43,000 from business operations dependent upon the system, which made erroneous calculations. If the developer were to successfully correct the system, the user would not be able to sue based on tort for the $43,000 in lost profit. The user would only receive an upgraded version or the purchase price, whichever the developer chooses to provide. Such a result is generally the case unless the defendant owes a duty to the plaintiff independent of the contract (Vancouver Furniture v. General Electric Retail Systems, 1992). It is possible that malfunctioning software may someday be subject to the additional theory of strict liability in tort, which imposes liability based on product defect. This is not yet the case (Miyaki, 1992).

Fraud

A cause of action for fraud generally requires that the plaintiff, seeking a recovery, prove (Analysts International v. Recycled Paper Products, 1987):
(1) that the defendant, made a statement,
(2) of a material nature,
(3) which was untrue or made in reckless ignorance of its truth or falsity,
(4) was relied upon by the plaintiff to his detriment,
(5) made for the purpose of inducing reliance, and
(6) that the reliance led to the damages.

The usual claim would be that the user was fraudulently induced to enter into the contract by the false representations of the developer, vendor, or consultant (RKB Enterprises v. Ernst & Young, 1992). A typical allegation of fraud is that the developer represented that the software could meet the user’s needs when it knew the software could not, or failed to disclose previous significant problems with the software. If upheld, the allegation of fraud in the inducement is separate from the breach of contract claim, possibly allowing a broader or larger recovery for the user (Schleifer, 1986). For example, in the software industry, developers have been aware of defects present in newly released versions of software as was the case with Intuit and Disney, but deliberately hold off releasing fixes and upgrades and may even deny the existence of problems. In such instances, would the tort of fraud apply? Would such nondisclosure be considered to be “of a material nature,” as specified in (2) above? The answer to these questions will become clearer as software related cases continue to be litigated, setting precedents and establishing patterns for interpreting the developer’s explanation for the presence of software flaws.

Negligence

The tort theory of negligence is applicable in cases within which four elements are present:

(1) the existence of a legal duty of care owed to the injured party,
(2) a breach of that duty,
(3) actual and proximate cause of the injury by the breach of that duty, and
(4) damages (Lyman, 1992).

Negligence applied to professionals is referred to as malpractice. Malpractice has been defined as “professional misconduct or unreasonable lack of skill... Failure of one rendering professional services to exercise that degree of skill and learning commonly applied under all the circumstances... with the result of injury, loss, or damage” (Black, 1979). Usually in malpractice actions, there is a professional licensing system, code of ethics, and formal disciplinary system (Condo, 1991). These elements are found in such occupations as law, medicine, and public accountancy, but are not currently enforceable within the software industry. The tort based theory of computer malpractice, that is, negligence applied to computer professionals, is still in its embryonic stages. Without a clearly determinable professional standard, courts have generally not allowed a tort action for computer malpractice (Wolpert, 1993).

Despite the general disallowance of a computer malpractice cause of action, there is judicial precedent supporting this theory. In Diversified Graphics, Ltd. v. Groves, the Court affirmed an $82,500 computer malpractice award, based its decision in part on the failure of the defendant, Ernst and Whinney, “to act reasonably in light of its superior knowledge and expertise in the area of computer systems.” It should be noted that in its interpretation, the Court discusses that Ernst and Whinney, acting as IS consultants and not as CPAs, incorporated in its Guidelines to Practice the AICPA’s (American Institute of Certified Public Accountants) “Management Advisory Services Practice Standards.” These general “standards” are meant to provide guidance to accounting firms, not for the development of computer software, and thus should not be used to establish a professional standard for IS services. This case is currently a distinct, but prominent, aberration.

Strict Liability in Tort

As software increasingly impacts more areas of business and our lives, the issue arises as to whether software should be treated as “products” and subject to strict liability in tort. The broad principle of modern product liability law developed primarily to spread the risk of physical injury resulting from manufactured products. As previously mentioned, software is not usually thought of as being associated with physical harm, but certainly can be. For example, in Sparacino v. Andover Controls Corp., an experienced chemistry teacher suffered acute bronchial damage, respiratory, and cardiac problems from chlorine gas when the laboratory exhaust fan failed to operate. The school district had installed a computer system that rendered the exhaust fan inoperable until 6:30 am. The teacher began his preparation for class at 6:00 am. There were no warnings or manual overrides. Should the software developer be held liable? The court decided it should not.

The courts have not yet been willing to apply strict liability in tort to software. Besides the issue of whether software is a “product” as previously discussed, there are policy reasons for not yet subjecting software to the rigors of liability under the tort of strict liability. Under this theory, liability is based upon defect, not fault (i.e., breach of duty). Since the software industry is still regarded to be a developing industry, it is arguable that the imposition of strict liability in tort would cause an undue financial burden. Further, it is argued that the imposition of this theory would have a chilling effect on the future of software development (Miyaki, 1992). Although the issue is undecided, the courts will ultimately have to decide whether large companies in the software industry, like Microsoft, should be subject to the same risk of liability as other product manufacturers.
Conclusion

Software developers themselves have expressed greater optimism regarding their own abilities to avoid flaws in systems development (Neumann, 1993). Yet, even software considered to be correctly programmed has had serious problems. NASA for example, experienced several difficulties in life-critical applications, including a synchronization problem with the backup computer before the first Columbia launch, multiple computer outages on a subsequent Columbia mission, output misreading that caused liquid oxygen to be drained off just before a scheduled Columbia launch, Discovery’s positioning problem in a laser-beam experiment, a reversal of shutdown procedures for two computers controlling Discovery, difficulties in an Endeavor rendezvous with Intelsat due to not quite equal values being equated, and others (Neumann, 1993). Indeed, “there are no guaranteed assurances that a given system will behave properly all of the time, or even at some particularly critical time” (Neumann, 1993). Nonetheless, software developers must continue to strive to develop reliable software or be held accountable by the courts for financial loss or physical harm resulting from software defects. Although the relatively few number of court actions brought against software developers may seem to imply the threat of legal action is remote, the recent changes in market perceptions may prove otherwise.

References


Apollo Group, Inc. v. Avnet, Inc. 58 F. 3d 477 (9th Cir. 1995).


Foster, Ed, “Software Won’t Ever be Perfect, But Can the UCC Provide Key Protection?” InfoWorld, February 26, 1996a, p. 54.


Patterson, Jere, (1990, June 4). “Say ‘No’ to Bugs,” Industry Week, 44.


There: The Movie Was a Hit, The CD-ROM a Dud; Software Bites
Schleifer, (1986). “Damage Awards and Computer Systems -
Smith, N. Craig (1995, Summer). “Marketing Strategies for the
Vancouver Furniture v. General Electric Retail Systems, WL
123829 (9th Cir. 1992).
Wolpert, Thomas (1993). Product Liability and Software Im-

Janice C. Sipior is an Associate Professor of Management Information Systems at Villanova University. She earned
her Ph.D. in Management Information Systems and her M.B.A. from the State University of New York at Buffalo,
and her B.S. from Canisius College in Buffalo, NY. Dr. Sipior has published in Communications of the ACM, Data
Base, Information and Management, Information Resources Management Journal, and numerous other academic
and professional publications. She is Vice-Chair of the Association for Computing Machinery - Special Interest
Group on Management Information Systems (ACM-SIGMIS) and was Editor of MIS Interfaces, the newsletter for
the Institute for Operations Research and the Management Sciences - College on Information Systems (INFORMS-
CIS) for six years. Her current research interests include ethical and legal aspects of information technology, system
development strategies, and expert systems.

Burke T. Ward is Chair and Associate Professor of the Business Law Department and a faculty member of The
Graduate Tax Program at Villanova University. He earned his LL.M. in Taxation from New York University Law
School, and his J.D. from Seton Hall University Law School. Dr. Ward has published numerous articles in the areas
of taxation and employment law. He is Director of Estate Planning for the law firm of Rainone and Rainone,
Philadelphia, PA. His current research interests include the legal aspects of information technology, estate
planning, and business succession planning.

William P. Wagner is an Assistant Professor of Management Information Systems at Villanova University. He
earned his Ph.D. in Management Information Systems from the University of Kentucky. Dr. Wagner has published in
research interests include knowledge acquisition for expert systems, international business MIS/DSS, and multi-
media.
Related Content

Examining Online Purchase Intentions in B2C E-Commerce: Testing an Integrated Model
[www.irma-international.org/chapter/examining-online-purchase-intentions-b2c/5519/](www.irma-international.org/chapter/examining-online-purchase-intentions-b2c/5519/)

A Context-Based Approach for Supporting Knowledge Work with Semantic Portals
[www.irma-international.org/chapter/context-based-approach-supporting-knowledge/10201/](www.irma-international.org/chapter/context-based-approach-supporting-knowledge/10201/)

A New Theory of Cognition and Software Implementations in Information Technology
[www.irma-international.org/article/new-theory-cognition-software-implementations/4138/](www.irma-international.org/article/new-theory-cognition-software-implementations/4138/)

The Post-Adoption Phase of Broadband in Small Businesses
[www.irma-international.org/article/post-adoption-phase-broadband-small/38909/](www.irma-international.org/article/post-adoption-phase-broadband-small/38909/)

Timing Strategies for Feasibility Studies in Information Systems Development
[www.irma-international.org/article/timing-strategies-feasibility-studies-information/50925/](www.irma-international.org/article/timing-strategies-feasibility-studies-information/50925/)