

# Software, Regulation, and Domain Specificity

William R. Bush

*December 2006*

See <http://dx.doi.org/10.1016/j.infsof.2006.08.007>

## **Abstract**

The growing pervasiveness of computer systems is bringing with it more societal reliance on those systems, which in turn is attracting the attention of various legal and political entities. This increasing attention will, one way or another, result in more regulation. This paper discusses regulation and its various forms, its effects on software development, and the software development tools and techniques that can be used to respond effectively to the demands of regulation. In particular, the maturing of software technology is leading to domain specific solutions that fit the needs of both software developers and regulators.

*Keywords:* Regulation of software; Domain specific software development; Realtime software; Web software development

## **1. Introduction**

There is a saying, “Be careful what you wish for -- you just might get it.” The computer industry and computer science community have prospered and become significant from the ubiquity of computers. With that ubiquity, however, has come growing reliance by society on the proper operation of computer systems. That, in turn, is leading to more and more oversight by legal and political institutions. Software and its development will have to more broadly adjust to this oversight. Making the adjustment more difficult are characteristics of software development that make it more of a legalistic process and less of a technical one. Fortunately, in some domains software is already subject to stringent oversight; those domains can serve as examples. Also, computer systems and their implementation techniques are maturing, and thus tailored, higher quality, domain specific solutions can be more naturally constructed.

This paper examines these issues, namely: the how (not the what) of the growing regulation of software, the similarities between software development and legal processes, and the evolution of new software development artifacts and techniques that will help developers deal with a more regulated future. The specific context of the legal analysis, which is meant for technologists and is suggestive, not exhaustive, is the United States.

This paper evolved from an appearance before a committee of the National Academies [1] investigating the development of dependable software, “Sufficient Evidence? Building Certifiably Dependable Software Systems” [2]. The author has a background in both law (a J.D. degree) and computer science (a Ph.D.), and has been involved with the Internet since 1970 (when it was the ARPANET).

## **2. Regulation and Software**

In general, regulatory forces impinge on an activity whenever society relies on the activity, the consequences of its failure are significant, and ex post facto remedies are inadequate. Thus building construction is regulated to promote fire safety, utilities are regulated to protect electric power and telephone service, and automobile construction is regulated to enhance occupant safety.

Some software has from its first deployment been critically important, and hence regulated, such as avionics software. On the other hand, much has been peripheral to society, such as desktop client software (or its precursor, time sharing). Now, though, as computers become pervasive, more and more software is becoming more and more significant and potentially subject to oversight.

This section briefly presents various general types of regulation and some significant existing regulations, and then considers the specific case of desktop client computing as an example of a traditionally unregulated class of software that is headed for more regulation.

## 2.1. Regulatory Paradigms

Various regulatory regimes can be imposed on software development. The following list is not meant to be precise or exhaustive, but rather suggestive. “Regulation” in this context broadly means the imposition of liability and constraints on software development that are external to the contractual relationship between buyer (user) and seller (developer).

One can identify four types of regulation, in increasing order of randomness (that is, relative lack of initial participation by the regulated and resulting lack of control by them over the outcome):

*Self regulation.* Industries have trade groups that also set standards for the members, or separate standards bodies. These organizations, while possessed of their own politics and struggles for competitive advantage, present a united front to the world at large.

*Government regulation through administrative proceedings.* Typically created by legislation, government agencies oversee the operation of an industry deemed to be important (or aspects of it). Such agencies follow standard administrative procedures that usually involve participation by the regulated companies.

*Government regulation through legislation.* Various legislative bodies pass laws compelling behavior, a process managed through lobbying and the media.

*Litigation.* Aggrieved parties bring lawsuits after the fact seeking monetary damages. Expensive judgments rendered in such suits can substantially affect the subsequent behavior of the defendants (and, more broadly, an industry).

When a group sees the mechanism of a lower level of regulation as inadequate (or if, for personal gain, it can portray regulation as inadequate) it attempts to raise the level. Catastrophes help cause escalation.

The factors that affect how, when, and what form of regulation will be imposed are numerous and varied. Consider:

*The cost of regulation.* It is easier to regulate a few large companies than many small ones. It is more rewarding to sue a large, wealthy company than a small one.

*International, national, and local concerns.* Regulation may be imposed at various levels. Regulation reflects various priorities, local conditions, and beliefs. In the United States, for example, building construction is regulated at the level of individual towns, counties, and states, whereas radio and television are regulated nationally.

*Resolving overlapping regulations.* Companies must deal with overlapping, and conflicting, requirements. For example, automobile manufacturers must deal with overlapping and conflicting safety and emissions requirements.

*The type of harm.* Regulation (especially litigation) is more likely and more costly when physical harm to people is possible. Thus in the U.S. there are the Federal Aviation Administration (FAA), the Food and Drug Administration (FDA), and the National Highway Traffic Safety Administration (NHTSA), and a legal industry pursuing lawsuits against airlines, drug companies, and automobile manufacturers.

*The degree of harm.* Often, below some level, activities are tolerated. For example, the Federal Communications Commission (FCC) allows unlicensed broadcasting on some frequencies below certain power thresholds.

*Regulation as a competitive weapon.* Companies will sometimes use regulation to their competitive advantage. For example, companies in the U.S. use the antitrust laws competitively.

*Indirect regulation.* Activities that are easy to regulate are constrained in order to control activities (typically behavior) that are hard to regulate. For example, requiring device manufacturers to implement copy protection is an indirect method of regulating consumer copying behavior. Lawrence Lessig believes that government will widely seek to use Internet software for indirect regulation [3].

Note that, in general, regulation has predictable side effects:

- Regulation increases the costs of the regulated.
- Regulation may cause activity to move to other, unregulated jurisdictions or to other, cheaper alternatives.
- Regulation creates a compliance industry. This consists of lawyers, trainers, and experts who advise clients and their staffs on how to comply with (or evade) the regulations. An example compliance service can be found at

[4].

- Regulation also creates a policing industry, composed of a combination of public and private sector organizations.

Examples of significant current regulations that affect software development and its deployment are:

- The Health Insurance Portability and Accountability Act of 1996 (HIPAA) [5]. This law mandates the use of standardized, electronic transmission of administrative and financial data between health care providers and insurance companies.
- Common Criteria [6]. This standard was jointly developed by the U.S., the U.K., Canada, France, Germany, and the Netherlands for the evaluation of computer security criteria. First released in 1983, the standard has evolved to cover various aspects of security and system development at various levels of rigor.
- FIPS PUB 140-2, Security Requirements for Cryptographic Modules [7]. This standard was developed by the U.S. government for the certification of devices that perform cryptographic functions. Both hardware and software are covered by this standard, which certifies devices at varying levels of rigor.
- The Sarbanes-Oxley Act of 2002 [8] [9]. This law imposes new rules of conduct on U.S. corporations, with respect to accounting, disclosure, and conflict of interest. It does not deal with computer systems, but compliance in fact requires substantial computer support.

These regulatory mechanisms effectively mandate different properties of computer systems:

*Defining requirements.* Common Criteria defines required security functional components.

*Requiring certain algorithms.* FIPS 140-2 requires the use of certain cryptographic algorithms.

*Requiring certain data formats and their content.* HIPAA (elaborated by rulemaking) specifies data formats and codes that are to be used in computer systems.

*Requiring the production of certain artifacts during development.* Common Criteria mandates the construction of specific artifacts during development to facilitate evaluation.

*Requiring standards of behavior of humans using computer systems.* HIPAA and its regulations define not just computer system functionality, but the procedures people using those systems should follow.

A catastrophe could accelerate developments or change their direction. In the U.S., 9/11 has energized the federal government. The Department of Homeland Security and the Department of Defense are both heavily involved in securing critical platforms, and have various initiatives underway.

Ultimately, regulation will make two demands on software: higher quality and assorted specific features that promote various regulatory goals (which may or may not have anything in the first instance to do with computers).

## 2.2. The Regulation of Client Computing

General desktop client computing technology has largely evolved free of externally imposed constraints. The constraints have instead fundamentally been technological and economic. This subsection examines the historical context of that evolution and how circumstances are changing. It informally addresses the questions of how and why traditionally unregulated software producers could be held liable for software problems.

To illuminate the evolution of software technology and its relationship to external forces, consider four simple classes of interested parties for a given software artifact (such as an operating system, programming language, database system, or media player):

*Users (or, in a business context, consumers).* Basically, they simply want a solution to a given problem.

*Technologists.* These are the research and development professionals who invent basic technology, implement prototypes, and often produce deployed artifacts. Their rewards are fundamentally intellectual.

*Monetizers.* These are the entrepreneurs, managers, and employees who generate and maintain commercial artifacts. Their rewards are primarily financial.

*Regulators.* These are policy makers, enforcers, and lawyers who represent the users and the larger world. Their rewards are political.

Note that one person can play multiple roles.

Now consider the evolution of client computing. The key is the change in the user community. There are three stages:

*The age of technology, 1960-1980.* This period saw the development of timesharing, personal workstations, Ethernet, and the ARPANET. Most network client computing was done as funded computer science research. Software was largely produced by technologists who were also the users. Thus almost all users were relatively sophisticated. Note that the core open source community formed during this stage, and is still basically composed of technologists with a base of technologist users.

*The age of commercialization, 1980-present.* During this period the IBM PC and the networked Unix workstation appeared. Subsequently Windows became ubiquitous and TCP/IP capable, and browsers enabled the web. All of these technologies had existed in some form before 1980, but had not emerged from the computer science community. It was during this period that fortunes were made by the monetizers. The key was broad use of computer technology by non-technologists. (See *Crossing the Chasm* [10], by Geoffrey Moore, for a general discussion of technology adoption).

*The age of regulation, present-future.* This period will see client software technology become a necessity of life, as the telephone and internal combustion engine have. With ubiquity comes dependence, and with dependence comes societal oversight, in order to guarantee reliability, among various other goals. Users will include people who resist and resent technology, and regulators will represent their interests.

Automobile technology provides a useful analogy. It too went through three stages. The first stage was largely experimental. The second stage was one of broad deployment enabled by mass production (specifically the Model T Ford). This deployment also caused great changes in infrastructure, in particular the development of a high quality system of roads. (With respect to client computing, the current building of broadband infrastructure is analogous to the development of the road system.) The current, third stage is one of dependence and concomitant ever-greater regulation. Technological development during this stage has been guided by regulation, and has included major safety and good citizen enhancements (seat belts, airbags, and fuel economy and clean air improvements). One can expect analogous safety and good citizen requirements to be made of client computing.

The issue of client software regulation sometimes arises in casual conversation in the form of a specific question: given all the reported security problems with Windows, why hasn't Microsoft been sued under U.S. product liability laws? The answer helps illustrate where client software development is in its regulatory evolution.

For a lawsuit to prevail (absent a specific new statute enabling liability) a court (and jury) would have to make three findings:

*The End User License Agreements (EULAs) limiting liability are unenforceable.* These contractual provisions, imposed by all software vendors before software can be used, severely limit liability. A court would have to void these provisions to find liability. Courts have the power and the legal doctrine to do this. "Bizarre or oppressive" terms in boilerplate contracts imposed by large corporations on consumers can be voided as adhesion contracts, and "unconscionable" terms that are "unreasonably favorable" to, and imposed by, one party on another can also be voided [11]. For example, in *Steele v. J.I. Case Co* [12], the court set aside a warranty provision that severely limited the liability of a combine manufacturer to a farmer, on the grounds that the differences in position between a powerful corporate seller, who dictated the warranty terms, and a private individual buyer were so great that enforcing the provision would be unfair and inequitable. Courts, however, rarely apply these theories, because they are loath to interfere with contracts generally, and will do so only when they perceive gross unfairness.

*Some action or role of Microsoft's makes it liable.* For example, it could be found negligent in its software development practices, which would, informally, have to be worse than average in the industry; that would be shown with a battle of expert witnesses (the outcome of which is often uncertain). Or an implicit warranty could be imposed on Microsoft, due to its role as supplier; it would be difficult, however, to show that courts should impose such a substantial burden.

*Damages are assessed that make the lawsuit worthwhile to the plaintiff's lawyers.* A monetary value would have to be put on the time lost (or other damage done) that would motivate lawyers to sue Microsoft. Losses due to problems with an individual computer would likely not be substantial enough, so potentially thousands of plaintiffs would need to be aggregated into a class action lawsuit and represented by one legal team.

Such a lawsuit is unlikely in the near future due to the following factors:

*Lack of physical harm.* In the U.S., juries respond generously to the display of physical harm suffered by plaintiffs (and inflicted by large, wealthy corporations, such as cigarette manufacturers, drug companies, and auto makers). Absent such harm, courts are less likely to impose liability. Legislatures are less likely to enact new statutes.

*Public attitude toward computer technology.* Computers are still new and exciting, to some extent, and vendors of computer products are seen as innovative and forward thinking. They have not yet been demonized. Microsoft in particular is in a strong position with its image marketing, its lobbying, and its promotion of its efforts to improve the quality of its software. Courts will be unwilling to penalize it.

*Lack of understanding of technology by lawyers.* Lawyers and the legal system are slow to deal with new technology. As long as large damage awards can be gotten in other, proven ways from wealthy companies, Microsoft is relatively safe.

As circumstances change, however, especially with respect to public attitudes, the liability climate will change as well.

The fact that the software industry is unlikely to soon become the next tobacco or asbestos industry does not mean that the legal system will not intrude.

Near term likely intrusions will be forced by:

*More contractual requirements.* Large customers are pressuring software suppliers to provide some indemnification for losses due to software failure. The unhappier big customers get, and the more bargaining power they get through alternatives, the more they will push for this.

*More standards compliance required for purchase or use.* Either by statute, by administrative finding, or as a matter of practice, buyers will be forced to buy systems that comply with various standards. See Common Criteria and HIPAA above. Compliance will also be used by sellers to control market access, as is the case now with, for example, ebXML, ROSETTA, and OASIS.

*More regulation of human activity, leading to software requirements.* Regulation of various activities will indirectly require computer systems to guarantee compliance. See Sarbanes-Oxley above.

For extended discussions of specific regulatory activity with respect to the Internet, see the books by Lawrence Lessig [3] and Stuart Biegel [13].

### **3. Software Development as a Legal Process**

It is notoriously difficult to develop and maintain large software artifacts. Paradigms from other disciplines have been applied to software development, such as architecture, mathematics, and biology. In fact, software development has much in common with the drafting and amendment of statutory law.

Consider:

*Design is often driven by examples, anecdotes, and scenarios.* Examples, etc., are used to motivate features.

*Design is driven by complex social realities.* There are conflicting requirements and stakeholders, each pushing to see certain features and using different examples.

*Artifacts are debugged by testing.* In the case of law, judges, via judicial review, are the initial debuggers.

*The older and more stable the artifact, the more reliable it is.* A consequence of debugging by testing is that the less tested an artifact is, the less well understood and less reliable it is.

*There is constant pressure to add features.* Various groups want their own problems solved, their own needs (often economically based) attended to. Law and software are never finished.

*Long term costs are hard to estimate and are often ignored.* In contrast to physical artifacts, it is relatively easy to add a feature (pass a law or write some code), but there are often large hidden long term costs. Traditional physical engineering, on the other hand, characterizes failure modes and prevents them within a budget.

*Artifacts are brittle.* Artifacts have complex internal and external relationships that can be easily disrupted by change.

*The longer an artifact has been around, the harder it is to change.* People make large investments in existing artifacts based on details of specific features, and resist incurring extra costs due to changes.

*The creators of artifacts are often not directly responsible for the consequences of their work.* The creators of law are insulated by political processes; the creators of software are protected by contracts (specifically EULAs).

*Artifacts embody the producing organizations and the history of development.* Since artifacts evolve largely in an ad hoc fashion, they embody that history.

*Patterns often guide the construction of artifacts.* Creators use general forms and specific details of other artifacts to create new ones and modify them.

On the other hand, software development is different from law:

*With software, there is a quasi-physical artifact, and it can be tested immediately.* Law operates indirectly on the world, by affecting human behavior, whereas software directly affects the behavior of computers. The larger software is and the more it interacts with humans, the more like law it is and the harder it is to test.

*The nature of tests and failures is different.* Law never crashes (absent general social upheaval and revolution, a fortunately rare event in industrial societies and beyond the scope of this paper), and can't be partially tested offline. Law doesn't really have the concept of an alpha or beta test.

*The law of unintended consequences is not so prevalent with software.* Ultimately the domain of law is all human behavior, whereas the domain of software is narrower (at least until strong AI exists) and more predictable.

*Dissembling and disingenuousness are not so overtly rewarded with software.* Law deals directly with human behavior, and money and power flow directly from it.

Despite these differences, the similarities to law are worrisome with respect to software, demonstrating why it is often of poor quality. In addition, there is one meta-problem with both law and software, to which many of these other problems contribute, and that is complexity.

Complexity is largely encouraged in both law and software (at least, significantly, in the short term):

*Features come first.* Features demonstrate observable activity and progress. Lawmakers are largely expected to be activists, and a new version of software requires something obviously new. Also, creators of artifacts, and their patrons, are by nature optimistic about the prospects for their creations and naturally underestimate problems, and thus put in more features than they should.

*When problems arise, there are strong pressures to rework rather than redesign (and simplify).* Both creators and customers become invested in understanding the existing system, and would rather that it be tinkered with than that it be improved.

*Both disciplines are relatively detail-intensive and theory-poor.* There aren't many theories to drive design choices, and there are many details to manage, so uncoordinated details pile up.

*Lawyers and software developers find complexity intellectually challenging.* It is stimulating to design something complicated, requiring powers of memory and concentration.

Software organizations further reward complexity:

*Mastering complexity is often rewarded institutionally.* Simple is equated with easy, even though designing an elegant solution is harder than hacking up a complicated one. Hacking up a quick solution that is never fully debugged is rewarded over producing an elegant solution more slowly.

*Promoting complexity is often rewarded commercially.* Features sell, and the more complex the feature set and interfaces the harder it is to copy the software's behavior.

This complexity is unfortunate, because it makes for more difficult debugging, which is harder (more expensive, more time consuming) than design and coding.

Fortunately there is a simplifying trend in software, which is discussed in the next section. Alas, there is no such trend in law. For example, the U.S. tax code [14] and its attendant rules runs to 66,000 pages [15]; over 15,000 changes have been made in the last 20 years, doubling its size. People would much rather keep (or get) a specific benefit than give one up for the generic cause of simplification [16].

Note that the ever greater power of computers has been used to build complex, feature-rich, relatively unreliable platforms, rather than simple, less efficient, more reliable ones. This is paralleled in law, where wealthy societies in general have more complex legal systems, requiring more lawyers and other professionals.

Also note that Lawrence Lessig has made an analogy between legal code and software code [3]. The heart of his analogy is that both codes control behavior. He also believes that open source software will substantially affect government regulation by making source code visible to its users. In fact, however, almost all users, including technologists, are unable to deal with the vast amount of source code that goes into platforms and applications. Thus the ultimate key to regulating code is controlling the distribution of binaries.

#### **4. Domain Specificity**

Regulation requires effective technological solutions. Regulation of software technology is based in part on societal reliance on that technology, but without effective solutions, long term reliance isn't possible.

The maturation of software technology has naturally led to experience with specific, focused solutions. The mature engineering knowledge embodied in these effective solutions is by its nature in part domain specific.

Thus the fundamental regulatory need for guaranteeing predictable, high quality results and the growing body of effective software solutions meet in the realm of domain specificity.

Furthermore, regulation is a two-way street. Lawyers and politicians are largely not capable of defining engineering-based rules. Those rules must fundamentally come from the regulated. They do so in the form of domain specific expertise. Further, the less explicit legal liability tests of negligence are based on reasonableness, and evidence of standard practice is highly relevant.

Thus the software technology of domain specificity -- programming languages, platforms, development tools, and development techniques tailored to specific problem domains -- should be examined generally and with respect to specific domains, to better understand how regulation has been and can be enforced with respect to software.

This section presents a general description of software and domain specificity, a look at domain specific aspects of the relatively mature field of realtime systems, and a view of domain specificity in the rapidly evolving area of web infrastructure.

The following section discusses technical directions that can be taken to improve the effectiveness of domain specificity with respect to regulation.

##### **4.1. Domain Specificity in General**

In U.S. patent law, scientific truths are not patentable, but processes and machines are. An initial attempt was made to patent a program (a BCD to binary conversion technique) as a "mathematical procedure", but the U.S. Supreme Court saw this as an attempt to patent a law of nature, and rejected the attempt [17]. Subsequently an attempt was made to patent a machine (a rubber curing device) that included software to control it [18]. The Court in this case held that the presence of software did not make the machine unpatentable, and allowed a patent to issue, thus beginning the modern era of software patents.

An analogous evolution in conceptualization can be seen with respect to software itself. As it has become more pervasive it has become more specialized, more bound to the hardware in which it is deployed and to the problems that it solves. Economically, software has become just a material component of a solution that must also involve hardware.

As software generally has become more specialized -- more domain specific -- it has evolved to embody domain specific knowledge and expertise. Even early languages and platforms had to serve some user community well to be successful; examples are FORTRAN (scientific computing), COBOL (business computing), and LISP (symbolic computing). These languages have continued to evolve to serve their domains, incorporating and modifying new ideas to fit their paradigms.

This evolving specialization can be seen clearly with the current versions of Microsoft's Windows [19]:

- client Windows: Windows XP (Home, Professional, Media Center, Tablet PC, Professional x64, Plus!);
- server Windows: Windows Server System;
- Windows for embedded and realtime systems: Windows Embedded (Windows CE 5.0);
- Windows for small, mobile devices: Windows Mobile.

This list also shows that large, specific individual domains are appearing, some of which will be discussed in detail below.

For software development, the main benefit of domain specificity is effective project maturity -- the more efficient and more predictable production of high quality artifacts. Rather than starting from a clean slate, existing, institutionalized knowledge can be leveraged immediately. The adaptation of general, relatively inefficient solutions can be avoided. The less new code that has to be written, the more quickly stability can be reached and, potentially, the simpler artifacts can be.

In addition, domain specificity also helps address some of the legal analogies of software development:

*Design is often driven by examples, anecdotes, and scenarios.* As much as possible, these examples, etc., have been abstracted into domain specific knowledge.

*There is constant pressure to add features.* Domain specific platforms tend to draw a clearer line around what is well supported (optimized), and what is not. This aids in the resolution of feature selection.

*Long term costs are hard to estimate and are often ignored.* Domain specificity helps ameliorate costs in general, and there is some accumulated domain specific cost expertise.

*Artifacts are debugged by testing,* and

*Artifacts are brittle,* and

*The longer an artifact has been around, the harder it is to change.* Domain specificity involves the use of specialized architectures that may require less testing and less modification to adapt to new circumstances.

*The creators of artifacts are often not directly responsible for the consequences of their work.* Domain specific development methodologies exist that hold developers and managers personally liable for their work.

*Patterns often guide the construction of artifacts.* Domain specific techniques are informed by specialized patterns of use.

#### **4.2. Realtime and Embedded Systems**

The domain of realtime and embedded systems has existed since computers escaped from large machine rooms and began controlling physical devices and processes. Compared to the traditional world of general purpose computing, the realtime domain has distinctly different requirements and practitioners, and has evolved a well-developed set of languages, platforms, and techniques.

Fundamental to realtime systems are their timely sensing of and responding to physical phenomena. This requires platforms that have sophisticated support for timing, deadlines, and scheduling, and that are concurrent, efficient, and predictable. In addition, realtime systems can be large and complex (managing, for example, ships or electric power stations), and often must be extremely reliable and safe [20]. Industries that develop realtime systems include aerospace, automotive, communications, consumer electronics, and industrial automation and process control.

As a result of their characteristics, realtime systems have been coping with regulation and liability from the beginning. Getting special attention have been safety critical systems, ones for which a system failure can cause loss of human life. Such systems include medical instruments, nuclear power stations, and aircraft.

The regulation of avionics software is instructive. Computer software in civilian aircraft must be certified in the U.S. by the FAA under the DO-178B standard, which was developed by the Radio Technical Commission for Aeronautics in the U.S. (RTCA) [21] and the European Organization for Civil Aviation Electronics (EUROCAE) in Europe (in Europe the standard is known as ED-12B). This standard does not require any specific software features, but instead defines various required software development practices, depending on the consequences of software failure. There are five levels of criticality, intuitively named: catastrophic, hazardous, major, minor, and no effect. Getting software certified at the highest level is so difficult that it naturally imposes severe constraints on the nature and complexity of the software. Rigorous documentation is required, documents must be signed by the responsible parties, and misrepresentation can result in personal, criminal liability.

A second class of realtime system is designated mission critical. These are systems for which failure does not result in loss of (civilian) life, but can cause a defined mission to fail. Software in the unmanned space probes of NASA's Jet Propulsion Laboratories and in virtually all military systems falls in this category. (Military systems are mission critical rather than safety critical because possible loss of life is an anticipated consequence of some

military missions.) Mission critical systems are not subject to the same rigorous certification regimes as safety critical ones, but have similar innate requirements.

The combination of strong reliability requirements, a relatively mature domain, and interaction with the physical world, has led to the development of relatively sophisticated and specialized realtime software development techniques and models of the physical world.

For example, realtime concerns with time and predictability have led to:

- specialized languages (such as Ada),
- specialized platforms (various version of realtime Linux, for example),
- specialized tools (compilers, debuggers, and analyzers), and
- specialized programming techniques (cyclic executives, rate-monotonic scheduling, coping with priority inversion, ...).

There is substantial and ongoing research in these areas (see, for example, the proceedings of the ACM Conference on Language, Compiler, and Tool Support for Embedded Systems).

But at least as important to the quality of realtime systems as the above software artifacts and techniques is effective physical modeling. Realtime systems' interactions with the real world are based on (implicit or explicit) models, and if those models are inaccurate the system will not operate properly.

Thus physical domain expertise is as important to realtime systems as software domain expertise. Knowledge of guidance and navigation, propulsion, aerodynamics, and control theory is at least as important to the construction of a spacecraft, for example, as knowledge of programming techniques.

In fact, the traditional method of developing realtime systems with extensive physical models (such as flight software or automotive engine control software) is to use two teams, one of physical domain experts who develop the models, and one of programmers who translate the models into code. In many organizations the domain experts are valued more highly than the programmers.

Over the years modeling languages and environments have emerged that support physical modeling by domain experts, the best known of which is MATLAB. These languages, environments, and toolsets have evolved substantially in sophistication (see, for example, MATLAB and Simulink [22], and LabVIEW [23]).

Significantly, these environments support programming activities and the automatic generation of code from models. This is important because it optimizes the connection between physical domain expertise and the production of software. This in turn promotes the improvements in software quality and simplicity that will be needed in the future.

### **4.3. The Evolving Web**

Unlike the realtime and embedded domain, the world-wide web is relatively new, is descended from desktop computing, and is not closely tied to the physical world. Nonetheless, because of its rapid acceptance and wide deployment, it has evolved quickly and is displaying some domain specificity.

In particular, it is rapidly evolving:

- specialized languages (especially scripting languages),
- specialized platforms (with a focus on media and mobility), and
- specialized tools (environments for web development).

In fact, the evolutionary nature of web software may obscure the distance covered in the last ten plus years. In the early to mid-1990s there were: desktop operating systems, simple browser programs, and a simple embeddable virtual machine. Now there are evolved and specialized operating systems for various mobile and consumer devices, browsers that are programmable content rendering and media delivery engines, and Java platforms and libraries that support smart cards, mobile phones, set-top boxes, desktops, and enterprise servers (there are in fact 146 standardized collections of targeted Java functionality known as JSRs [24]).

A second wave of specialization has appeared, building on the first, and enabled by simple browser enhancements (dynamically replacing a portion of a page, changing styles on the fly, and fetching content from a server in the background without a user click event). Ajax, for example, builds on existing browser (JavaScript) and web (XML) technology to create interactive web applications [25]. Another well-known example is Ruby on Rails

(RoR) [26], a web application framework written in the Ruby scripting language that supports the quick generation of applications that access networked relational databases. (There is also Ajax on Rails, which combines Ajax and RoR.) Yet another example is Adobe Flex [27], a framework for building Flash enabled applications.

A third wave will probably appear soon, in which browsers will move even further from their HTML roots and become, via ubiquitous JavaScript, more capable UI clients with widgets and full control over user events.

All of this evolving platform specificity is a big step toward mature domain specificity.

Also, significantly, ever more sophisticated development environments have appeared for web developers (Dreamweaver [28] and FrontPage [29], for example). These environments support the necessary features of web applications: the design of the appearance of pages (partly by providing templates), the use of web data formats (specifically XML), the use of standard web protocols (FTP, WebDAV, SSL), and the writing of code using web-centric languages (HTML, CSS, PHP, JavaScript).

As can be seen from the features of these environments, web domain experts -- web developers -- must be graphic designers and business process experts; they enable the commercial essence of the web. They must also understand networking issues and new media formats. Programming per se is secondary.

The big, ugly, looming problem with all of this is quality.

The W3C HTML validator tool [30] informally demonstrates an aspect of this. Virtually no commercial website has valid HTML (for example, the Amazon site generates 1144 errors, eBay 248, google 61, and Elsevier 3).

More formally, a recent check of a collection of the 10,000 most popular websites found that only 5% had completely valid HTML [31]. This situation was due to

- authoring tools that generate non-standard HTML,
- browsers that accept non-standard HTML, and
- the importance of look and feel in websites over correctness.

It is in the interests of browser writers to handle as many web pages as possible, including badly formed ones. It is in the interests of the suppliers of authoring tools to produce good looking web pages, especially when those suppliers also make dominant browsers that handle special nonstandard appearance features. And it is in the interests of web developers to build attractive websites quickly, even if that means copying other sites that themselves were poorly constructed. Regulation is exactly the force that gets applied in such circumstances to change everyone's interests.

A problem that results from these factors is that web page hackery leads to browser complexity and hackery, which lead to insecurity. A study [32] [33] uncovered at least one remotely exploitable flaw in every browser tested, and more than 50 flaws in Internet Explorer, a handful of which could be used to gain control of the user's system. Microsoft's response was instructive: the flaws were "stability issues and not security vulnerabilities." From a security standpoint this is disingenuous, since every flaw is a potential vulnerability. But from a long-term evolutionary standpoint it is true that browsers, one way or another, will (have to) become better defined and more stable.

As the web becomes another key infrastructure component, as relied on as the telephone, higher quality will simply be necessary, and will be mandated in one legal form or another: contracts, standards, administrative rules, legislation, or litigation. The likely motivation will be security. Every bug is an exploit, and hackers are just testers with different test cases and a different agenda.

As mentioned above, regulation of aspects of the web other than quality is already here. This regulation will grow, and some of the regulations will affect software development (with respect to, for example, requirements, algorithms, data formats, and protocols).

## **5. Strategies for Software Development**

Forces arrayed against software developers are a lurking government and legal system, and legalistic development practices. On the other hand, the growing maturity of computer systems is helping offset those forces with domain specific solutions, which can be constructed more directly and more simply, and can ultimately be of higher quality.

A number of research efforts and products specifically demonstrate how software development should cope with these forces.

*Simplified domain specific platforms.* Domain specific platforms and their benefits were discussed above. A significant additional benefit is relative simplicity. A platform that has been implemented from the outset for a particular domain can be simpler than a specialized version of a general one. Examples of this include: the Spotless virtual machine [34], which became the KVM [35] and was used to build a FIPS-140-capable Java implementation [36], and a Java platform for safety critical systems [37], an ongoing standardization and commercialization effort.

*Built-in domain specific security.* Domain specific security mechanisms can be built in to a platform, which will generally make them simpler and more secure than general purpose ones. Compare, for example, the security mechanisms in [36] with those in [38].

*Domain specific analysis tools.* Analysis tools can be made more effective by focusing on domain specific problems in selected code. This is especially valuable with analysis techniques that are either expensive per se or expensive in the general case. Examples include SLAM [39] [40] and PREfast [41] [42] for Microsoft device drivers.

*Domain specific construction tools.* As described above, domain specific environments with appropriate templates and libraries already exist, and can be made more sophisticated. But beyond these, tools are being created that allow developers to specify systems at a higher level, leveraging the efficiencies of greater abstraction along with the benefits of formal derivation of running code. These abstract approaches are difficult to make work in general, but can be optimized for specific domains. For example, the AutoSmart system generates correct smart card applications from high level specifications [43].

*Domain specific construction tools for domain experts.* As described above, physical domain experts already use models to design and build embedded systems; automatic error checking of the code generated from these models has been introduced [44]. More work can be done in this area, to enable more focused domain specific modeling, and to automatically produce optimized implementations. This will eliminate steps and potential errors in the realization of systems. An example of this work is the MDL project [45], which takes JPL's MDS language [46], used for specifying state-based realtime systems, into realtime Java.

*Tool support for certification.* Current certification programs (such as Common Criteria, FIPS 140, and DO-178B) impose extensive documentation burdens on the software development process. Tools can potentially lighten these burdens: enhanced analysis tools can produce reports to augment or supplant test results, and tools that derive code from models and specifications can trace (or even prove) connections between high and low level artifacts.

Many of the above aids come from tools. In general, although software is often developed in ways similar to law, it is ultimately used in a physical system and is thus produced with tools. Tools provide functions that are missing in the law. They provide artifact based abstractions, measurements, controls, and contexts.

In the limit case, a feature simply requires that there exist an execution path that exercises the feature. On the other hand, ultimate quality demands that for every path through the code certain results do not occur (and that certain results do occur for some paths). As tools and techniques become more domain specific and more sophisticated, they are evolving to strike a balance between the rapidity of feature development and the thoroughness of exhaustive analysis.

The U.S. government, recognizing the importance of tools, is beginning an effort to evaluate them. The SAMATE (Software Assurance Metrics and Tool Evaluation) project is managed by NIST's Information Technology Laboratory [47]. Supported by the DHS, the overall goal is the identification, enhancement, and development of software assurance tools, with the specific goals of: testing software evaluation tools, measuring the effectiveness of tools, and identifying gaps in tools and methods. In this context, software assurance means:

- trustworthiness -- no exploitable vulnerabilities exist, either of malicious or unintentional origin, and
- predictable execution -- justifiable confidence that software, when executed, functions as intended.

The project is already accumulating test cases for tool evaluation. Ultimately this work could evolve into tool evaluation criteria, which will lead to de facto standards via purchasing.

Another consequence of maturity, domain specificity, and regulation is the ever greater specialization of software developers. Defining the appearance of a web page is a fundamentally different activity from modeling the chemistry of an internal combustion engine. Both of these activities result in software but require vastly different skills and knowledge. Regulation may come to recognize these differences in experience, requiring that personnel be certified to perform certain tasks. It may also emphasize differences in degree of experience as well as kind, perhaps requiring certification of senior engineers and mandating their oversight on any significant project.

## 6. Conclusions

Enabled by the microprocessor, computing has become practically ubiquitous. As computer systems become just another everyday artifact like the telephone or the automobile, politicians and lawyers will assert ever more control over the technology. This control will take the form of statutes, rules, and liability through lawsuits, regulating the development, deployment, and use of computer systems. It will have as its target both computer technology per se and the human behavior enabled by that technology.

The computer industry can head off some of this regulation by improving quality, regulating itself via standards, and collaborating with government regulators. There is no immediate prospect for liability of the type seen imposed on drug or tobacco companies, but it is hard to predict what will happen in a generation or two; familiarity breeds contempt.

In coping with regulation, software development will naturally struggle with its own legalistic tendencies. In particular, complex, feature rich, quality poor, hard to maintain systems will be problematic.

Fortunately, as computer systems are maturing they are naturally becoming more specialized and solution specific, and software is slowly morphing into a component of a solution rather than a stand-alone technology. This domain specificity, most visible in embedded and realtime computing, but rapidly developing with Internet software, will help developers deal with greater demands from regulators.

Domain specific platforms, languages, tools, and techniques are leading to greater simplicity than general solutions, embodiment of solution oriented knowledge, and more intimate involvement of non-programming domain experts. These in turn lead to higher quality, more responsive development, and greater developer effectiveness. In the future, domain specific tools in particular will play an important role. As software development becomes more regulated, specialized tool support will be necessary to facilitate compliance.

## 7. Acknowledgments

I would like to thank Randy Enger, Bernard Horan, Ellen Peel, Gilbert Whittemore, Mario Wolczko, Alessandro Coglio, and the anonymous reviewers for many useful suggestions, and my daughter, Elizabeth, for being patient.

## References

- [1] The National Academies; <http://www.nationalacademies.org>
- [2] Project CSTB-L-02-07-A; <http://www8.nationalacademies.org/cp/projectview.aspx?key=235>
- [3] The collaborative effort to update CODE and other laws of cyberspace; <http://codebook.jot.com/WikiHome>
- [4] Cybertrust: Compliance/Governance; [http://www.cybertrust.com/solutions/compliance\\_governance/](http://www.cybertrust.com/solutions/compliance_governance/)
- [5] HHS - Office for Civil Rights - HIPAA; <http://www.hhs.gov/ocr/hipaa/>
- [6] Common Criteria Evaluation and Validation Scheme; <http://niap.bahialab.com/cc-scheme/>
- [7] FIPS PUB 140-2: Security Requirements for Cryptographic Modules; <http://csrc.nist.gov/cryptval/140-2.htm>
- [8] H.R. 3763; <http://news.findlaw.com/cnn/docs/gwbush/sarbanesoxley072302.pdf>
- [9] American Institute of Certified Public Accountants: Sarbanes-Oxley -- The Basics; <http://cpcf.aicpa.org/Resources/Sarbanes+Oxley/Sarbanes-Oxley+%E2%80%93+The+Basics.htm>
- [10] *Crossing the Chasm*; Geoffrey A. Moore; Collins Business Essentials; 2002.
- [11] Electronic Commerce: On-line Contract Issues; [http://www.oikoumene.com/ec\\_contracts.html#adhesion](http://www.oikoumene.com/ec_contracts.html#adhesion)
- [12] *Steele v. J. I. Case Co.*, 419 P.2d 902 (1966).
- [13] *Beyond Our Control? Confronting the Limits of Our Legal System in the Age of Cyberspace*; Stuart Biegel; MIT Press; 2001.
- [14] U.S. Tax Code On-Line (as of January 2004); <http://www.fourmilab.ch/uscode/26usc>
- [15] Taxing Our Patience; Senator Olympia Snowe; <http://snowe.senate.gov/ws4-21-06.htm>
- [16] "An Approach to Income Tax Simplification"; Fred W. Peel; *University of Arkansas at Little Rock Law Journal*, Volume 1, Number 1; 1978; pp. 1-38.
- [17] *Gottschalk v. Benson*, 409 U.S. 63 (1972).

- [18] Diamond v. Diehr, 450 U.S. 175 (1981).
- [19] Microsoft Windows Family Home Page; <http://www.microsoft.com/windows/default.msp>
- [20] *Real-Time Systems and Programming Languages, 3rd Edition*; A. Burns and A.J. Wellings; Addison-Wesley; 2001.
- [21] Radio Technical Commission for Aeronautics; <http://www.rtca.org/>
- [22] The MathWorks - Products and Services; <http://www.mathworks.com/products>
- [23] LabVIEW - The Software That Powers Virtual Instrumentation - National Instruments; <http://www.ni.com/labview/>
- [24] The Java Community Process Program; <http://www.jcp.org>
- [25] *Pragmatic Ajax: A Web 2.0 Primer*; Justin Gehtland, Ben Galbraith, Dion Almaer; Pragmatic Bookshelf; 2006.
- [26] Web development that doesn't hurt; <http://www.rubyonrails.org>
- [27] Adobe Flex 2; <http://www.adobe.com/products/flex/productinfo/overview/>
- [28] Macromedia - Dreamweaver; <http://www.adobe.com/products/dreamweaver/>
- [29] FrontPage Product Overview; <http://www.microsoft.com/office/frontpage/prodinfo/overview.msp>
- [30] The W3C markup validation service; <http://validator.w3.org>
- [31] "An Experimental Study on Validation Problems with Existing HTML Webpages"; Shan Chen, Dan Hong, Vincent Y. Shen; *WWW 2005*.
- [32] "Browsers feel the fuzz"; Robert Lemos; SecurityFocus 2006-04-12; [http://www.theregister.co.uk/2006/04/13/data\\_fuzzing/](http://www.theregister.co.uk/2006/04/13/data_fuzzing/)
- [33] The Metasploit Project; <http://www.metasploit.com/>
- [34] "The Spotless System: Implementing a Java System for the Palm Connected Organizer"; Antero Taivalsaari, Bill Bush, Doug Simon; *Sun Microsystems Laboratories Report TR-99-73*; February 1999.
- [35] The K virtual machine (KVM); <http://java.sun.com/products/cldc/wp/>
- [36] "A Mechanism for Secure, Fine-Grained Dynamic Provisioning of Applications on Small Devices"; William R. Bush, Antony Ng, Doug Simon, Bernd Mathiske; *Proceedings of the 2004 Workshop on the Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS04): Lecture Notes in Computer Science*, Volume 3362 / 2005; Springer Verlag; January 2005; pp. 86-107.
- [37] *Concurrent and Real-Time Programming in Java*; Andy Wellings; Wiley; 2004; Ravenscar Java, Chapter 17, pp. 347-373.
- [38] *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*; Li Gong, Gary W. Ellison, Mary Dageforde; Addison-Wesley Professional; 2003.
- [39] The SLAM Project; <http://research.microsoft.com/slam/>
- [40] "Microsoft's Secret Bug Squasher"; Simson Garfinkel; <http://www.wired.com/news/technology/bugs/0,2924,69375,00.html>
- [41] PREfast for Drivers; <http://www.microsoft.com/whdc/devtools/tools/PREfast.msp>
- [42] "PREfast: Less [sic] Bugs, More Reliability"; Stephanie Horstmanshof, Suzanne Ross; <http://research.microsoft.com/displayarticle.aspx?id=634>
- [43] High-Assurance Java Card Applets; <http://www.kestrel.edu/jcapplets>
- [44] PolySpace Collaborates with The MathWorks and Launches PolySpace for Model Based Design; [http://www.polyspace.com/pr/2006-04-04\\_Mathworks.htm](http://www.polyspace.com/pr/2006-04-04_Mathworks.htm)
- [45] "NASA's Exploration Agenda and Capability Engineering"; Daniel E. Cooke, Matt Barry, Michael Lowry, Cordell Green; *IEEE Computer*; January 2006; pp. 63-73.
- [46] "Engineering Complex Embedded Systems with State Analysis and the Mission Data System"; Michel D. Ingham, Robert D. Rasmussen, Matthew B. Bennett, Alex C. Moncada; *American Institute of Aeronautics and Astronautics First Intelligent Systems Technical Conference*; September 20-22, 2004.
- [47] Software Assurance Metrics and Tool Evaluation; [http://samate.nist.gov/index.php/Main\\_Page](http://samate.nist.gov/index.php/Main_Page)

All web citations were current as of 10 August 2006.