International Journal of Information Technology Education 1(1): 115-128.

What Makes Software Special – and Especially Faulty

Jonathan P. Caulkins
Carnegie Mellon University

Abstract. Software failures are a bane of modern existence. Information systems deliver phenomenal functionality with appalling lack of reliability. (Unfavorable) comparisons are drawn with manufactured goods. This article suggests that is the wrong metaphor. For purposes of understanding reliability, software is better compared to design goods, like scripts, architectural plans, and consulting reports. Design and software development are both diverse, so it is not the case that every design activity is like every software project, but for any given software effort, a design activity that matches on key dimensions (extent of customization, scale, backward compatibility constraints, etc.) may be a useful foil. In general, reliability of engineered systems, like airplanes, may be a better referent for software quality than is manufacturing process control.

Keywords: software, quality, reliability.

Acknowledgements: Ashish Arora, Fusun Gonul, and Rahul Telang contributed many insightful comments to the development of this paper.

1. Introduction

Software unreliability creates substantial costs for customers and for society more generally. Some think software's unreliability is not noticeably different than for comparable non-software products and/or that, even if it were noticeably different, there are no grounds for treating it differently in a legal or policy sense. E.g., existing standards for holding software developers liable are fine and do not need to be adjusted. Others think that software is systematically less reliable than it should be for reasons that are traceable to unique characteristics of software and which may warrant some sort of software-specific intervention or regulation to improve social welfare.

To date the surrounding debate might aptly be characterized as the clash of the bromides: "There is nothing new under the sun" vs. "the information revolution changes everything," with individual's positions tending to be highly correlated with the institutional interests of their employer.

This paper strives to concisely summarize arguments for software being different in a balanced fashion. Not surprisingly the conclusion is that both

"sides" in the debate have valid points. Software is different, but those differences do not render insights and experiences from other domains wholly irrelevant.

2. Software Reliability is Different Than Hardware Reliability

When one discusses whether software is different, it is important to be explicit about what it may or may not be different than. Few would argue with the proposition that software is different than a horse, but a horse is not a relevant foil.

A common referent is non-software products, particularly manufactured goods. Later we will suggest that in important respects design activities may be a more appropriate comparison, but initially we will discuss how software development is different from producing bolts and cars and bowling balls.

2.1. Software Failure is More Likely to Be a Discontinuous Function of Its Inputs

Software failures are often sudden and unpredictable. The software works fine one moment, but the next moment it fails, when to outward appearances nothing of any consequence has changed. Consider in contrast, car failures. Often an engine makes unusual noises for some time before a failure becomes so serious that the car simply will not run. Likewise, when spark plugs get old or a cylinder begins to lose compression, a car will lose its pep rather than switch instantly from perfect working order to frozen.

As with all aspects in which software appears to be special, there are counter-examples. A thrashing CPU has both slow response time and vulnerability to freezing. Simple objects like bolts and fasteners or pressurized vessels, including tires, may fail catastrophically without warning – or at least seem to do so if they are not inspected frequently and carefully. (And even careful inspection may not be informative for certain composite materials such as those used in the tail fins of some Airbus aircraft.)

Nevertheless, at the level of a generalization, there is a difference. Indeed, the fact that sudden failures in cars are often associated with electronics (e.g., a short circuit or failed ignition solenoid) not mechanicals is instructive.

Mathematically, chaos is associated with a nonlinear relationship between inputs and outputs. Software is worse than chaotic; the output is not just nonlinear, it can actually be a discontinuous function because it is built on logic gates. In contrast, the physical world and physical products are, at least at human scales, customarily modeled as being continuous. The industrial age

rested on calculus and Newtonian mechanics. The information age rests on discrete mathematics.

This distinction between software and physical products has implications for the ability of testing programs to be reassuring in terms of risk of subsequent failures. If a car engine starts at 10 degrees Fahrenheit and at 30 degrees, that's a good basis for predicting that it will also start at 20 degrees. If it starts easily at 10 degrees, one expects it to start – albeit perhaps with some difficulty – at 0 degrees. But such modest extrapolation, and even interpolation, is less reliable with software. That a software system works with input set #1 and input set #3 does not guarantee that it will work with input set #2.

One might cry foul at this example, pointing out that temperature is a continuous measure whereas the input set numbering is cardinal and arbitrary. But in some sense that is the point. Many of the environmental inputs relevant to failure for a car or other physical product – temperature, load (in the mechanical engineering sense of the word), speed, etc. – are continuous. Some are for software as well, such as utilization and file size, but many are not.

2.2. Software Does Not Wear Out

In a literal sense, software almost never "wears out". If the software was able to execute some calculations correctly when it was new, it generally will also do so on the 100^{th} or the $1{,}000{,}000^{th}$ replication of the identical task. Not so for a car. The day it is purchased, (hopefully) it starts on the first try. One expects the car to likewise start on the $1{,}000^{th}$ try, but not the $10{,}000^{th}$, at least not without replacing worn parts along the way.

Indeed, for physical systems, quality and reliability are often conceived of specifically in terms of resistance to wear failure. Companies boast that their products are "built to last" and the Energizer Bunny's battery "keeps going and going".

Software products do not necessarily have a longer productive life than do physical products. I drive a 1990 Prizm, but am typing in Windows 2000. However, my copy of Windows 98 did not wear out; it just became obsolete, which is different, at least in some respects.

Software is more like milk than tires. Software, like milk, will "go bad" within a certain amount of time even if I do not use it heavily. In contrast, at least if protected from ozone and sunlight, tires will keep their tread and pliability almost indefinitely if they are not used. Software is perishable, like milk, but it does not "wear out" the way tires do.

2.3. Software Has Minimal Quality Variability Over a Production Run

Production of digital products not only has near zero marginal costs, it also has near perfect conformance. In contrast, the entire science of statistical process control exists precisely because there is variability in the essential attributes of successive products coming off a given physical production line. Even for a product as simple as a ball bearing, diameters vary over a product run. In contrast, it usually makes literally no difference which copy of Madden's NFL2003 one picks off the shelf at BestBuy. Indeed, language is instructive. We speak of "copies" of software, but not of manufactured goods.

As a result, if one knows for sure that one copy of a software product works, or does not work, for a particular task, one can be very confident that another copy will perform similarly. In contrast, when doing a home improvement project, it is perfectly reasonable to look at each 2"X4" individually to see if one but not another has knots where you plan to nail or will allow a more attractive grain to be exposed to view. Likewise, one 2003 Peugeot can be a lemon, even if others, produced at the same assembly plant, are not.

2.4. "Tight Coupling" with Other Products that is Both Invisible and Managed by Laypeople

To function fully, software products often must be "tightly coupled" to other products. For example, my PC's operating system has to be able to talk to my printer and various application programs I load. These objects are distinct products made by different companies and bought at different times from different vendors. If they do not mesh together properly, there is a loss of functionality, in short, a failure. Indeed, with distressing frequency at home I find myself copying files to my wife's PC. For some reason still partially mysterious to me, her PC talks to her printer more successfully than mine talks to my printer.

Consider in contrast the coupling between a stove, a frying pan, and potatoes in that pan. It is not that all of my potato slices are exactly the same size and shape or even that I always use the nice Revereware pan instead of the cheap one picked up at a garage sale. Maybe one burner is more likely than another to burn the potatoes a bit, but there are no catastrophic failures. My stove has never refused to heat the pan when it contained Russet instead of Yukon Gold potatoes.

Obviously there are tightly coupled mechanical systems, a "Swiss watch" being the paradigmatic example. What seems different with software, however, is the frequency with which lay people must tightly couple objects whose interface is both complex and invisible. Anyone might wear a Swiss

watch, but only skilled artisans are expected to assemble, troubleshoot, or repair them. Furthermore, with physical systems, one can often literally see linkages. When my wife's presta-valve bicycle pump did not fit my schrader-valve tires, a moment's inspection revealed a round peg in a square hole problem. Computer peripheral manufacturers can give visual clues – mouse cords whose connectors are a different color and prong configuration than keyboard connectors. Some connectors even give a palpable and audible "click" when they seat properly. Software coupling is not similarly tangible.

An additional consideration concerns whether the attributes of the products being coupled that must mesh precisely are static or vary over time. When traveling overseas, my electric razor needs an adaptor to couple properly with an electric outlet. It has two flat prongs. I need three circular prongs in Europe and three flat ones in Australia. Not only can I see the difference, but neither the number of prongs on my razor nor the number of holes in the electric socket changes when I turn the razor on or move the voltage dial from 110V to 220V. In contrast, the settings that must mesh on my PC and printer include soft settings that can be modified by another program. So my PC and printer can be coupled effectively at one moment, and then stop talking when another program changes those settings. (E.g., when my word processor reads a file that was created by a German version of Word running on Linux, I cannot subsequently print an "All-American" file, even from another application.)

Hence, there are three distinct and at least partially distinctive characteristics of tight coupling with software: (1) lay people are expected to manage the coupling, (2) the coupling attributes are invisible, and (3) the coupling attributes are not stable over time. The combination of the three can generate "failures" for the user even if each software component is doing its job correctly, at least in the narrow sense of the word.

3. Software Errors Are Like Design Not Manufacturing Flaws

Because of these considerations, it is fair to say that software reliability is quite different than reliability in the context of manufacturing or other forms of mass production of physical goods. That does not imply, however, that software is unique with respect to considerations of reliability. Other analogies may be more constructive. In particular, when comparing software reliability to the reliability of other products, it is more constructive to draw analogies with design flaws in those other products, not with manufacturing defects.

Design flaws in physical products are not uncommon. There are pitchers that dribble, tricycles that are inordinately hard to assemble, and chairs that are simply uncomfortable. Often they manifest in abnormal operating circumstances (automobiles that are not protective in crashes), as the product wears out (de Havilland Comet jet crashes due to metal fatigue around its

square windows), or in failures under extreme conditions (O-rings on the space shuttle Challenger's solid rocket boosters failing in cold weather).[1] In contrast, software failures often occur during routine operations with "mint condition" software. Nevertheless, in both instances flaws stem from design and are built into every copy or unit produced, not just a subset that are "lemons". GM pickups burned in crashes not because sometimes UAW workers were careless, but because the pickup's design placed the gas tanks outside the truck frame's side rails.[2]

If one accepts that the better metaphor for software defects is design not manufacturing, it is not obvious why the metaphor need be restricted to the design of objects that will eventually be manufactured. That is, the failure patterns for software may resemble more generally that of other creative products, such as movie scripts, architectural plans, and consulting reports, as well as designs for manufactured goods. In all cases the quality problems are built into the design of the first (and sometimes only) copy, not introduced by the duplication process. And wear is not a primary consideration (although perishability may be). Failure for these other creative products may or may be so obviously discontinuous. If focus groups and pre-screeners like a movie immensely, one would not expect it to bomb with the general public.[3] Software design in that sense is more like a proof in mathematics. Introduce one false axiom, and the whole proof is worthless, whereas one forced metaphor in a novel does not make it impossible for the reader to enjoy other passages or chapters.

We will defer for the moment details of what kinds of design activities in particular make the best metaphor for understanding software reliability, and first discuss a few general implications of the design vs. manufacturing metaphor distinction. To clarify, by design we mean both instances in which a customer literally buys just the design (e.g., geodesic dome house designs can be purchased at http://www.aidomes.com/) and when the design is embedded in a physical product but the failings of the product stem from the first copy design, not its mass manufacture. For example, a murder mystery that gives away the ending in Chapter 1 might be a flawed product, but it is not the fault of the printing press. Likewise, in rejecting the manufacturing metaphor, it is important to stress that it is the idea of quality problems

---

1. See the Rogers Commission (1986) *Report of the Presidential Commission on the Space Shuttle Challenger Accident*. Washington, DC: US Government Printing Office.
2. Not all design flaws are the fault of designers; in this case the engineers seem to have been aware of the dangers from the outset but cost consideration drove management to opt for the cheaper but more dangerous design.
3. However, a movie can do very well with domestic audiences and decidedly less well internationally (How the Grinch Stole Christmas), or vice versa (Se7en), or play well to some demographic groups and not others.

associated with mass reproduction that is being rejected. Custom fabrication of single copies, which is often integrated with design, is still a relevant foil.

First and foremost, this distinction between design and (mass-)production immediately recalibrates expectations for quality. It is in manufacturing not product design that the six-sigma quality movement is promoted.

The modern quality movement in manufacturing strives to eliminate rework. The mantra is build it right the first time; don't inspect quality in ex post. In contrast, a common image of the elite artist or designer is of someone who perpetually writes and then rips up drafts as faulty. The quality standard depicted by the play (and movie) *Amadeus* (Mozart writing complete and perfect scores in his first draft with no erasing) was intentionally unrealistic; it was part of how the play (movie) depicted Mozart's work as more closely resembling the hand of God than a mortal's labors.

A second implication of design rather than mass production being the better metaphor for software creation is the notion that flaws come from disjunctions between specifications and the customer's desires, as well as from disjunctions between the specifications and the realized product. When thinking about quality vis a vis the manufacturing process, as opposed to product design, one focuses only on the latter.

Considering different specific types of software flaws and their analog in design and manufacture helps make the point. For concreteness, let the design object foils be the architectural plans for a building and a management consulting strategy report.

In software, a low-level flaw could be something like a typo that prevents code from compiling. In a consulting report it would likewise be a typo. An architectural analog would be specifying a component (e.g., a bolt) by a product number that did not correspond to any one's current parts catalog.

Another level of bug is internally inconsistent logic. An architectural metaphor might be dimensions that do not add up (e.g., opposite walls of a rectangular room being of unequal length). In management consulting it could be return-on-investment calculations for different options calculated with different discount rates.

The next level of flaw would be instances in which the program/design is internally consistent, but it does not meet the specification. The computer program was supposed to handle several cases, but can only handle one, although it does that one correctly. The building's roof was supposed to handle a 50-ton load of snow, but the design adapted from southern buildings would collapse under 40-ton loads. The client wanted a definitive quantitative basis for making a decision, but the consulting report only quantifies a subset of the relevant considerations.

At these three levels there are analogous flaws in manufacturing: chips, flaws, or impurities in materials; two parts of the assembly that do not fit

together; and parts whose dimensions vary from specifications by more than the allowed tolerance are typical examples.

There are three higher levels of software flaws (flaws at least from the customer's perspective). Their analogs in design are sufficiently obvious so not to warrant elaboration. In contrast, they have no analog when thinking about manufacturing process reliability.

These levels are: (1) Gaps between what the seller contracted to deliver and what was set down in the technical specifications given to the design team, (2) Gaps between what the customer wanted and what was contracted for, and (3) Instances in which the product does what the customer asked for at the time, but it does not perform well when the customer tries to use it for some other purpose, a purpose that was not anticipated at the time the order was placed. Both the first and the second typically pertain to problems of translation between higher-level descriptions of a product ("We'll build you efficient order-management system") and detailed contractual language. The distinction between the two is essentially who is legally responsible for making the translation correctly. In the first case, it is the developer; in the the buyer. But if one views successful product development as delivering something the customer values, not just delivering something that meets the letter of the contract, the distinction blurs. Collectively the two levels account for many serious software failures and much dissatisfaction about other design products, including architectural plans and consulting analyses.

The third level (problems arising when software is used for something other than that for which it is designed) is also responsible for many failures in the user's eyes, although from a developer's perspective they can with some justification argue that these are not flaws in their product. Still, given the frequency with which code is re-used and the rate at which technology is changing, it may be that software is used more often for things for which it was not designed, than for things for which it was. Furthermore, even if these failures are not design flaws per se, they are clearly instances of lack of quality. When we praise designs (software or otherwise) for "withstanding the test of time" we are recognizing that robustness is an integral part of truly good design.

To recap, the point here is that in thinking about manufacturing failures, one thinks primarily about conformance to given specifications. When discussing design failures, we consider not only parallel issues but also higher-level issues. Understanding the customer's needs and creating specifications is an integral part of the value creation process; failure to do so is a type of design error. In this sense, design is a better metaphor for software flaws than are manufacturing flaws.

This second idea stemming from the design vs. manufacturing distinction could be summarized as: With design, failure is defined in part by a design's relationship to the customer, not just to the specifications. For all but the

simplest designs, design failures also often have to do with the relationship between the design object and other objects that are not part of the design. Design quality is contextual and less purely an intrinsic attribute of the object in isolation. The same tends to be true for software, in part because of the "tight coupling" discussed above. This can make defining failure more subjective, or at least more dependent on knowledge of the context. There are software failures that look like mechanical failures, e.g., the system crashes and becomes inoperative. Often, however, failures take the form of output that does not make semantic sense or that triggers problems when coupled with some other component.

4. Software and Design: Parallels and Contrasts

Design is a very broad domain of human activity and so might seem not to be a very useful metaphor for software creation and its reliability. However, software products are likewise diverse and many of the distinctions between types of design activities have counterparts in distinctions between types of software development. Furthermore, when the counterparts on one side or the other do not exist, that is itself instructive. For instance some types of designs can have important aspects of their quality checked by visual inspection, e.g., with scale architectural models. That software analogs do not jump to mind is a reflection of points raised above in the context of tight coupling between components. That distinction does not mean all design is different than software development; for some designs, such as complex circuit diagrams, it is hard to judge quality by mere inspection.

Important distinctions among different design activities, and the analogous distinctions for software, include the following.

4.1. Is the design customized to a specific customer or developed for the prototypical customer among many?

A strategic consulting report is tailored to a single paying client in much the same way that customized software is. There may be commonality across reports a single consulting group gives to different clients, just as custom software applications for multiple clients may reuse common code, but they are nonetheless customized. In contrast, there are millions of paying customers for a hit movie, a SuperMario brothers game, and Microsoft office suite.

Quality issues are different for customized vs. mass-market software and design. As discussed, customized products have quality issues related to

conformance with specifications and whether the specifications as written were in fact what the client wanted written. Software and design products developed for a mass market have to contend more with whether they are appealing enough to be purchased, create customer satisfaction, and generate referrals. Failing to please 95% of anonymous customers in a general market is as much a failure as is failing to please the one customer for a customized project, although it is less tangibly so because people who choose not to buy something are less of an injured party, or at least are less vocal, than is someone who agreed to buy something that was never delivered. When comparing quality of one effort to another, it is most helpful if both are similar with
respect to this customized vs. mass market distinction.

4.2. How Mature is the Industry?

Reliability of all sorts of products improves over time, so even if it were true that personal computer software is less reliable than cars or planes today, a more relevant comparison might be to cars in 1915 or planes in 1925.

4.3. What is the scale of effort (no more than a few person years, or is it a large team effort)?

A novel or blueprint for a house is usually the product of a single person's creative genius. Consulting reports and plans for commercial buildings are jet fighter involves hundreds of person-years of effort. Quality control challenges vary with scale. Software development efforts likewise vary in scale by many orders of magnitude, in terms of persons involved and person-years of effort, with parallel implications for the difficulty of ensuring reliability. Simply put, coordination is difficult and a common source of errors. Some dissatisfaction with software quality may stem from comparing the quality struggles of software developed by large groups with the product of individual efforts (blueprints for a house) not the work of similarly sized teams (jet fighter designs).

Similar points could be made with respect to the size or complexity of the object itself. It is not clear how many lines of code are equivalent to one moving part in a mechanical system, but for any plausible "conversion factor", many computer programs have a complexity that is more on a par with a car or jet plane, or even an entire production line, than with most typical consumer goods.

4.4. Does the Customer Modify the Product or Use It As Is?

Jazz scores and cooking recipes are designs whose creators anticipate that the user will modify that design. Likewise, the success of a consulting report at generating profits for the client depends substantially on what the client does and does not do to implement the recommendations. In contrast, novelists do not expect readers to adapt their book, and a movie-goer need contribute little more to the artistic enterprise than to show up and stay awake. The reader and movie-goer may judge the work's quality but do not per se contribute much to it. Similarly for the average user of Microsoft Word, and end users do not generally tamper with the software in embedded systems, such as anti-lock brakes. Indeed, customers of such software typically do not even gain access to source code. In contrast, larger organizations acquiring customized software often "maintain" it with in-house staff adapting it over time, e.g., to interact with new peripherals. Inasmuch as Excel and Access are more platforms on which customers build their own databases and spreadsheet applications, Excel and Access are somewhere in between.

The more involved the user is in adapting the design, the harder it is for the vendor to make the design foolproof. If a cook deviates from the recipe and the soufflé falls, is it the recipe-writer's fault?

4.5. How Complex Are the System's Interactions with Its Environment

Many failures occur at the interface between systems. If the environmental interactions a system must support are highly structured and limited in number, there will tend to be fewer and more graceful failures. Likewise one would expect fewer failures if the surrounding environment is intelligent and adaptive. Laws, which are in a way design products, interact in more complex ways with their environment than do novels, and are more prone to unintended adverse consequences. Fortunately, however, the environment within which laws operate is populated with intelligent and adaptive agents (judges, administrative bureaucrats, etc.), so inane consequences of literal interpretations are minimized.

Some software has only highly structured interaction with its environment. My son has a small computer toy with just a few buttons. The possible set of inputs is very limited. Likewise, the only output device is a small screen. Input and output for typical computer games and some embedded software ABS brakes) is more complicated but still manageable and well defined. In contrast, a computer's operating system has to be able to respond to an immense array of conditions and inputs, and not all of its environment is intelligent and adaptive. (One can debate whether users are intelligent and

adaptive, but standard peripherals certainly are not.) It should not be surprising if operating systems crash more often.

4.6. How Important Is Backward Compatibility with Existing Systems?

Designs for a house addition have to be compatible with the existing structure; characteristics of the existing structure can greatly constrain the new design, forcing compromises that would not otherwise be made. The same can happen with laws and movie sequels. For the design of many physical products, however, the lack of "tight coupling" can leave one with a relatively free hand. That railroad car wheels have had a certain axel width means that tracks are also of that width and it would be very expensive to introduce a different width. Likewise, 35 millimeter cameras have to use 35 millimeter film if they going to be compatible with existing film processing equipment, but if someone develops a better design for a lawn mower, fishing reel, or jet engine, tradition and precedent are not likely to be overwhelming barriers to adoption. Software probably has more backward compatibility constraints, with some detrimental effects on quality. E.g., new versions of the Windows operating system persist in having long-known fundamental flaws in their security against Internet attacks. Presumably that is in no small part because fixing those problems could jeopardize compatibility with legacy software created on and for previous versions.

4.7. Do We Expect the Design to Function or Just Be?

It is easier to make a dish than a dishwasher not only because the dishwasher has more parts but also because the dishwasher is supposed to *do* something, not just sit passively. At some level all software does something. It converts inputs into outputs. Even screensavers are more active than books. Some non-software design objects do things as well. Decision rules (e.g., policies) convert circumstances into choices. They are a bit like a computer program in that regard, although the "environment" with which they interact is intelligent, at least if one thinks of managers as intelligent. However, many design objects are not in that sense functional. In this respect, first-copy designs of physical products embodied in working prototypes are a better analogy for software than are other information products such as architectural blueprints.

More distinctions like this could be drawn, but these suffice. Their collective point is that both design and software-creation are diverse activities. Hence, not every design activity is a strong parallel for every software activity. However, for any given software activity, if one found a design activity that

matched with respect to these attributes, it could be a useful foil for understanding reliability and other quality issues. And, when thinking about why one type of software might seem more reliable than another, the difference may stem from differences with respect to one or more of these distinctions.

For some types of software creation that are notoriously faulty, one is hard pressed to identify design activities with the same attributes that are highly reliable and successful. Consider Microsoft Windows. It is a mass-distributed shrink-wrapped design developed by thousands of people under severe backward compatibility constraints and that must function while supporting complex interactions with its environment. It is not clear that there is any design product outside the software domain with that combination of attributes. In some sense there is essentially no basis – outside of software – for projecting what the reliability of Windows "ought" to be and, hence, whether it is doing better or worse than should be expected.

5. Summary

There is widespread dissatisfaction with software quality. In particular, its reliability is often compared unfavorably with common physical products. This article suggests two broad reasons why software may fare badly in such comparisons. First, the relevant quality concept is with design, not mass production of physical goods. Software flaws are more comparable to design flaws than they are to production flaws. Second, both software creation and design are diverse, with a broad range of activities represented across a half-dozen or more dimensions that affect the frequency of flaws. Comparing software with a non-software design product that is unlike on one or more of these dimensions can be as misleading as comparing software flaws to manufacturing flaws.

The goal of this paper is to reduce the frequency with which inappropriate comparisons are made in unthinking ways, not to "solve" the software quality problem. Nevertheless, it does at least suggest what quality ideas might be the most fruitful to build upon. In particular, the analysis in this paper suggests that the total quality movement, statistical process control, and related ideas, however useful they may be in manufacturing, are not likely to be extremely helpful in the software domain, at least not without significant adaptation.

Instead, the efforts in other domains that seem more relevant would be design of complex one-of-a-kind or low-volume production systems such as planes, rockets, and factory assembly lines. So it may be that methods and insights from systems engineering and integration, e.g., as practiced in the aerospace industry, could be relevant. It is perhaps not surprising, in this regard, that there is so much overlap between defense contractors and large-

scale software developers. A question for further investigation is whether "pure" software houses populated by computer scientists and information systems experts but not project engineers of the aerospace variety would do well to import ideas from systems integration firms. Parallel questions for universities include whether computer science and information systems programs ought to require more courses in systems engineering, project management and the like, and whether techno-MBA programs preparing software industry managers ought to adapt their curricula to put more emphasis on the lessons learned over time by organizations such as NASA, Lockheed- Martin, and Boeing.