ARTICLE

On system design

JIM WALDO, Sun Microsystems, Santa Clara, CA, United States

Published: 16 October 2006

Citation in BibTeX format

OOPSLA06: ACM SIGPLAN Object Oriented Programming Systems and Applications Conference
October 22 - 26, 2006
Oregon, Portland, USA

Conference Sponsors:
SIGPLAN

OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (October 2006)
https://doi.org/10.1145/1167473.1167513
ISBN: 1595933484

On System Design

Jim Waldo
Sun Microsystems, Inc
1 Network Drive
Burlington, MA 01803
1 781 442 0497

Jim.waldo@sun.com

Abstract
In this essay, I consider some of the factors that are making it more and more difficult to expend the effort necessary to do system design. Because of changes in the economics of the field in both industry and research, we have become less able to take the time needed to do real system design, and to train the next generation of designers. Because of the intellectual property landscape, we are less able to discuss system design. The end result is that we do less good system design than we used to, at least in those environments where system design used to be most common. But there are reasons to be optimistic about the future of system design, which appears to be happening in non-traditional ways and in non-traditional venues.

Categories and Subject Descriptors D.2.2 [**Design Tools and Techniques],** D.2.11 **[Software Architecture]**

General Terms Design, management, standardization.

Keywords System Design, Education, Training.

1. Introduction

I am beginning to believe that the art and craft of system design is in danger of being lost. Carefully designed systems, in which the right abstractions are combined in just the right way to produce a system that is easy to learn, easy to change, and pleasing to use and work with, are unlikely to happen using the kind of design techniques that are popular today. It isn't just the techniques that we use that impede our ability to design systems. We are unable to train engineers and scientists adequately in system design. The economics of the industry push us in directions that don't favor design. The realities of funding in research make it unlikely that much time will be spent on system design. The end result is that less careful design work is being done, and we as an industry, a profession, and an intellectual discipline don't seem to care or be able to do much about it.

In what follows, I will try to describe and explain some of these factors, and try to make clear the price that the industry and the discipline are likely to pay because of these factors. I will begin by trying to characterize what we mean by system design. On the characterization I will give, all but the most trivial of software artifacts have a design, but only some of them were given that design consciously. I will then turn to how system design is learned, and given that as a base will look at the changes in both industry and academia that have made it harder for system design to be taught or even done in a reasonable way.

The inability to do or to learn system design in these traditional venues has led to the emergence of new areas where engineers and scientists can practice and perfect their skills in the area. I will end the essay by discussing some of those areas, as they provide the hope that good system design will continue to be part of what we teach, learn, and practice.

2. What Is System Design?

One of the most interesting, and most difficult, of the tasks that we may undertake in our careers as engineers or computer scientists is the design of an entire system. A system is a set of interacting parts, generally too large to be built by a single person, designed for some particular purpose. We work with systems all the time. The operating systems that control our machines are systems. The layers of hardware and software that allow the programs on these machines to interact with each other over a network are systems. Even most applications that we use are systems, whether we know it or not.

As engineers, we know that the way to solve a large problem is to break it into a set of interacting smaller problems. Each of these smaller problems can then be decomposed into even smaller problems, until (after enough iterations) we have a problem that can be solved on its own. System design is a similar task, taking a large system and breaking it down into a set of smaller systems. The additional step in system design is to specify the interactions of the smaller systems so that they fit together to create the larger system; after breaking the problem into smaller problems the system designer will say how it is that the solutions to those smaller problems will fit back together to solve the larger problem.

All (reasonable[1]) software is a system that has a design on this characterization. The software will be organized around methods (or procedures, or functions, or whatever abstraction for this sort of thing is supported by the language being used) and each of these methods will represent a decomposition and abstraction of a problem that must be solved for the software to run. The larger the piece of software, the more layers there are in the design, and the more complex the system.

[1] We have all seen BASIC programs that have no such design. However, after the first week or so of writing code, anyone who does not avoid that sort of non-structure should be quietly, but firmly, convinced to spend his or her time doing something, anything, else.

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-348-4/06/0010.

But to say that all software has a design does not entail that all software is designed. For a program to be designed requires that there be some (perhaps considerable) thinking about the right way to decompose the functionality, and how to create a small set of abstractions that can be re-used and re-combined to provide that functionality. The notion that anything that shows some kind of design was therefore the result of some conscious activity of design is a confusion that is the result of an ambiguity in the term "design." On one sense of the word, it is a property of some object (such as a program, a system, or the like) that merely indicates that there are parts that interact. On another sense of the word, it indicates the activity of determining what the parts of some larger whole should be, and how those parts will fit together. While anything that is the result of the activity of design will itself have a design, it does not follow that anything that can be described as a group of parts interacting to form a more complex whole (that is, anything that has the property of a design) is therefore the result of the activity of design.[2]

One of the best indications that a program is the result of the activity of design is the existence of a document that describes that design. But all too often software has a design that must be discovered from the code by inspection. Sometimes the design that is discovered shows all the hallmarks of a thoughtful design activity, but there are other times that the discovered design shows a haphazard combination of various abstractions, duplication of functionality in slightly different forms, and inconsistencies in the way in which abstractions were selected, implemented, and used. Such discovered designs show either the absence of any design activity prior to the construction of the program, or that what design activity did occur prior to the writing of the program was, to speak plainly, not very good.

Collections of programs that are meant to form a system exhibit the same sort of design, but at a larger scale and in a more public fashion. A collection of software meant to form a system will present the user of the system with a set of abstractions that can be combined in various ways. The abstractions may show symmetry, simplicity, and an aptness to the task that are characteristics of what we consider good design; the design may also seem random in its choice of abstractions, be repetitive, or show little consistency from one part of the system to another; these are systems that we consider examples of poor design.

I know of no adequate set of necessary and sufficient conditions for determining whether a design is good or not, but like so many things having to do with taste and aesthetics we generally know good (and bad) design when we see it. The Unix operating system has a simplicity and symmetry that is indicative of a good design; the companion C programming language has a combination of power and simplicity that both reflects and compliments the PDP-11 architecture for which it was originally intended.

There is no necessary connection between a system or program exhibiting good design and that system or program having been consciously designed by someone. While I personally know of no systems that were not actively designed that are examples of good design, there is no logical impossibility of such a system existing. Further, it is not all that hard to find systems that were consciously designed that are not examples of what any of us (other than, perhaps, the original designers) would consider examples of good design; I'm sure we can all come up with many examples of this sort of system. In spite of this, there does seem to be some connection between the activity of system design and the production of elegant, well crafted systems, in at least that all of the actual examples of the latter are also examples of the former.

System design can change and evolve over time. The original Java[tm] programming language and associated libraries had a simple and consistent design. Some of the additions to the libraries associated with the environment since it was first introduced reflect the original design, but others have introduced other notions of design. The overall system has evolved into something that, at a certain scope, has a coherent design but which, taken as a whole, is far less coherent than it once was. A more radical example of design change over time is seen in the sets of protocols and languages that define the World Wide Web; when first introduced these were simple and had a coherent design. Designs that have been proposed (or that have become accepted standards) in the past decade, however, show no such coherence and simplicity; individual collections of these may be said to form a designed system, but the amalgam that forms the overall platform does not.

Some of these examples of good design were thought out in fairly complete detail before the systems were produced. Others evolved with the implementation of the system itself. But in all of the cases of good design, there is a fairly simple set of principles that can be seen to underlie the design. In the case of Unix, the idea of a file and the ability of any program to take an ASCII stream as input and produce such a stream as output allowed those learning the system to know what to expect as they encountered new parts of the system. There are times that the design of a system, even when it is an example of good design, will need to be pushed and prodded in unnatural ways to gain something that the original design did not take into account. It was a great simplification in Unix to treat all files as ASCII streams, but the introduction of magic numbers, various kinds of headers, and conventions having to do with the filename extension show the desire for a typed file system being overlaid on such a system. While the original design of untyped files was simple and sufficient, the complexity that has grown up around the various ways of indicating the type of a file can make one wish that the original design had been somewhat more complex. But one should be careful about one's wishes; the number of bad designs that have been justified by quoting Einstein's maxim "everything should be made as simple as possible, but no simpler" makes one wish that he had actually said "everything should be made as simple as possible, and then made simpler."

Given my characterization of system design, I should really restate my concern on the subject. Since any system will have a design, saying that system design is dying out would be the same as saying that software development is dying out. That is demonstrably not the case. More and more software is being produced, so there are more and more system designs.

What I am worried about is the demise of systems that are designed, in the sense that there is some coherent plan for the system that is arrived at by the people working on the system in a way that is separate from simply observing how the code falls out. Maybe a

[2] The assumption that anything that exhibits the property of having a design is, therefore, the result of the activity of design is the base of arguments that go far beyond computer science, engineering, or programming. Space limitations and good sense keep me from addressing the wider issues of this debate.

better characterization of my worry is that the act of designing a system is happening less and less, and as a result the design of the systems that we are producing is becoming more and more haphazard and the resulting designs are less and less coherent, simple, and aesthetically pleasing. Certainly all of our software has some structure to it and some set of abstractions that can be identified as underlying the structure of the code, but I find that it is less and less common that the structure and the abstractions are thought about (and cared about) as entities in themselves. Instead, we seem to be producing software where the overall design can only be determined after the fact, by looking at the code that is produced

Whether the perceived lack of designing systems is good or bad, it is something that we as an industry and as an intellectual discipline should understand. The change in the design of systems is, I think, being caused by a number of factors. Individually, they might not be a problem; taken together they are changing the way we build systems. Part of it has to do with education; part if it has to do with economics; part of it has to do with the current fads or fashions in the way we write software. In what follows, I will look at each of these factors in turn.

Let's start with some thoughts on education.

3. TRAINING IN SYSTEM DESIGN

Like most other industrial research laboratories, Sun Labs brings in groups of interns over the summer to work on various projects. This is about as classic a win-win situation as can be found in business. The interns (most of whom are graduate students, but also undergraduates and the occasional high-school student) find a summer job in their field of interest, and the lab gets an injection of enthusiasm that is hard to replicate. The students think they are being overpaid, while we get what we consider cheap labor. The students don't know what things can't be done, and therefore often do the seemingly impossible. Best of all, they get to see what the "real world" is like (although the thought of an industrial research lab being a part of the "real world" is more an indication of how artificial the world of academia is than how real our world is).

This past summer, while walking back from lunch about a week into his tenure, the intern working in my group turned to me and asked, "So, how do you go about learning to design a system?" Like most great questions, it showed a level of naivety that was breathtaking. The only short answer I could give was, essentially, that you learned how to design a system by designing systems and finding out what works and what doesn't work. I've been thinking about the long answer ever since; I'm not sure that the long answer differs from the short answer in much more than length, but nonetheless here is what I've come up with.

3.1 The Origin of Good Design

Before knowing how to train someone in system design, it is useful to have some idea concerning the origin of good design. If we can know what leads to good design, we can try to teach people to do those sorts of things in the hope (although not the guarantee) that good design will result.

There is no shortage of books, seminars, and other training guides that claim to help in this quest. There are techniques (such as Six Sigma) that profess to aid in the development of good design. There are languages (such as UML) that claim to help in the development of good design. And there are no end to the methodologies and processes that claim to enable any team to create a good design that will meet the needs of the customer by the mere repeated application of the rules that make up the methodology.

I have no doubt that the success stories that each of these design approaches and aids cite are true. In some sense, that is just the problem; completely incompatible and contradictory approaches to the design problem have been shown to be wildly successful (by their proponents) and wildly unsuccessful (by the proponents of competing approaches). Bottom-up or top-down, waterfall or extreme; all seem to work for some and not for others.

The only generally applicable rule that doesn't have obvious counterexamples is one I first heard enunciated by Fred Brooks more than a dozen years ago. In a talk given in a Sun-internal seminar (an expanded version of which became the basis for his Turing Award lecture in 2000[3]), Brooks talked of the work he had been doing to try to find the underlying common feature of good design, not just in computer hardware and software but also in such endeavors as architecture, graphics, and the fine arts. The only thing that he could find that good designs had in common was that they were produced by good designers.[3]

There is one reading of this insight on which it is true but uninteresting, a mere tautological statement that reflects giving in to the unpredictable and inscrutable mystery of design. On this reading, the only way to determine what produces a good design is to wait until you have one, and then attribute it to the designer. Good design, on this view, happens by chance. You can hope for it, but you can't do anything to improve your chances of getting a good design.

This is not the reading that I believe Brooks intended, nor the one that I found persuasive when I first heard the talk. My reading of this principal is that those who have been able to produce a good design in the past are far more likely (although not guaranteed) to be able to produce a good design in the future. But there is no magic process by which such designers produce their designs; each may go about the design problem in a different way, and a designer may approach one problem in a particular way and another in a completely different fashion.

The point, I believe, is that good design is a capability that some people have, and others simply do not. Whether this is an innate skill that people are born with, or one that is cultivated over time in ways that we don't understand, is a question far too deep for me to address here. I neither know nor care. But by the time someone is designing a computer system, whatever it takes to be a good designer is either there or it is not. When it is there, it can be developed and honed (or, unfortunately, degraded and warped). But when it is not there, there is no technique or process that can make up the deficit.

There are a number of people who are uncomfortable with this concept. Many of them are managers; I will discuss their discomfort later. Others are uncomfortable with this view on more philosophical grounds; they feel that saying that there are those who can produce good designs and those who cannot is contrary to some egalitarian notion (which it is) and somehow elitist or undemocratic (which I think it is not).

---

[3] As a somewhat depressing side-note, the first question asked at the end of the talk (by a senior engineer) was what process those good designers used to produce the good designs. Sometimes hearing is not the same as understanding.

Why should we be surprised to find that there are some people who are just not capable of doing first-rate system design? Such designs are difficult, complex, and require a great deal of taste to get right. Further, they require the ability to deal with a great deal of ambiguity while forming the design, an ability to deal with whole sets of questions that can't be solved but which the system designer knows (or has the faith to believe) will be solved at the appropriate time. Given the difficulty of all of these tasks, it is no more surprising that not everyone can be a great designer than it is that not everyone can be a great composer, or a great artist, or a great architect (all fields that also require design). This is not to say that designers are better people than those who are not great designers; indeed, designers are good or bad people in roughly the same proportion as any other group. But it is to say that some people are better designers than others, and ignoring that is one of the many things that leads to bad system design.

3.2 Teaching by Doing

Having said all that, the question of how to teach system design is still open. The fact that good designs come from good designers does not tell us where the good designers come from. While it may be true that not everyone can be a good designer, it is also true that there is some learning that goes on. I am reminded of posters I saw years ago at the Rhode Island School of Design, posters with the headline "Talent without technique is a waste." The school did not claim to be able to make anyone an artist. But they did (and do) claim to be able to take someone with the talent to be an artist and give them the technique that will let them exploit and channel that talent. The same is true in system design; it may be that you have to have some talent to do the design task well, but it is also true that you need to learn the technique that allows you to channel and amplify that talent.

In my own case, the instruction that I received in system design came in the form of an apprenticeship with a master designer. This was not a formal arrangement, and it could well be that the person I considered myself apprenticed to did not see our relationship in anything like those terms. But looking back on it, I clearly see it that way.

The more structured and corporate relationship was that of an overall software architect for a major component of a system and an individual contributor for that system. The group I was in was responsible for the user environment component (basically, the windowing system and all user-visible tools) for Apollo Computer, an early workstation company. The architect of the group had implemented the first version of these components on his own, but grander plans had been hatched for the second system (as is always the case) and a small group had been assembled to do the design and implementation. I had been hired to design and implement the component library that would deal with text; there were others who were dealing with the windowing system, input mechanisms, the shell interpreter, and even a scripting language.

The overall design process for the group required the owner of each component to write a series of specifications for his or her component, starting with a straw man (a quick sketch of the various pieces and the overall component model) and ending with an iron man (which would be a detailed specification of all of the entry points and their functionality). Once a month, the entire group would go off site (usually to the apartment of the manager of the group) for a morning and review one of the specifications for some component.

The overall architect of the group was not one of the more active participants in these discussions. But when he talked, everyone else listened. His most damning criticism was a simple "That's too hard." When said of a specification, it indicated that you had not done the work to sufficiently understand the problem and boil it down to some simple core. The assumption was that there was always some simple core, and by making the assumption such a core was generally found.

These design reviews, and the constant interaction with both the architect and the other members of the group over a multi-year period of time, were the places where my system design skills were honed. It was here I learned about simplicity and symmetry, about interfaces and designing for change, and a host of other rules and techniques that I still use. More important, I learned what worked for me and what did not, and that what worked for me might not work for others. Rather than learning a process of design, I learned how I could best design.

I had originally thought that this way of learning design was unusual, and caused (in my case) because my academic background was in a field unrelated to computer science. But as I learned more and began talking to others who I considered to be good at system design, I found that this experience was more than just common; it was nearly universal. Everyone I talked to had a similar story of the master designer who had, either consciously or by example (and correction) taught him or her what they considered to be the important lessons in design. There was a period when I would ask, "who did you do your design apprenticeship with?" without supplying any other context. I expected some to be confused by the question, but I found that everyone to whom I asked the question not only understood it, but was able to answer without thinking. Even more interesting, the names that were given were often the same. Whether they knew it or not, a relatively small number of master craftsmen were credited with training a much larger number of system designers.

This was hardly a scientific survey, and as scientists we should take care in drawing strong conclusions from anecdotal data. But I think it is indicative of something that no one that I have talked to about how they design and how they learned to design has pointed to a class that they took which trained them in any important ways. Design, if my experience is any indication, is best learned by a long and varied process of trying, failing, and trying again under the guidance of someone who is an expert at the task.

3.3 Design and Curriculum

That no one seems to learn system design from some course can be troubling. If designing of systems is really the hard part of what we as engineers and computer scientists do, aren't we in need of some systematized way of teaching what is needed to do that kind of design?

Looking around the web, there are some courses in system design that are taught at various universities, and lots of courses offered by consulting companies. I have more than just a passing interest in a course in system design for a variety of reasons, not the least of which is that I have been contemplating teaching such a course. It is the sort of course that students ask for; it would be valuable if students coming into industry actually had some skill in system design; and it would be interesting to design the curriculum and readings for such a course.

I had great difficulty in getting anything like a set of readings or a coherent plan for such a class. There are some obvious readings (such as Lampson[6], and Brooks[2], and lots of things by Parnas[4]), but deciding on the concepts that needed to be taught and the sequence in which those concepts are to be presented keep eluding me. After enough time of trying and failing to come to some plan, I realized that I was thinking about this problem in the wrong way.

More than half a century ago, the philosopher Gilbert Ryle made a distinction between knowing *how* and knowing *that*[8]. Knowing that is a relation between a person and a proposition; it is a piece of factual knowledge that can be discovered, can be justified, and can be taught by the usual mechanisms of pedagogy. Knowing how is a different kind of thing; it is the kind of knowledge we have when we know how to walk, or run, or sing. It is not a factual sort of knowledge, but an ability that we exhibit in our actions. We can know how to do something reasonably well or expertly (while we can't know that the world is round reasonably well or expertly). Most important, while we can be taught to know how to do something, the kind of teaching that takes place is very different from the kind of teaching required to know that.

Academic disciplines require a combination of knowing how and knowing that. To be fully educated in any of these disciplines, one certainly needs to understand the factual backgrounds of that discipline. But to be truly educated in the field also requires that one learn how to think in a particular way. Each field has its own technique (or set of techniques) that must be learned just as well as the subject matter of the field if you really want to be an expert in that field.

Different fields have different combinations of subject matter (knowing that) and technique (knowing how). The vast majority of my formal (academic) training was in the field of philosophy; as practiced in the United States and England (the so-called "Anglo-American" or analytic approach to philosophy) the field is almost entirely technique. Certainly there is plenty of content (the history of philosophy and the great philosophical questions) but what really matters is the way in which one thinks (conceptual analysis, the building of logical models, approaches to argumentation). While very little of the subject matter of philosophy was useful to me when I became a software engineer, I found that the techniques I learned were just as relevant in computer science as they were in the field in which I learned them.

I'm told by those who have attended that law school is very much the same, in that gaining a technique (learning to think like a lawyer) is far more important than the actual subject matter of the law (which, after all, varies widely from locale to locale). After one has learned the technique, one can take the bar exam for a particular state (which tests knowledge of the subject matter of the law for that state) before one can practice law. But knowing the law without knowing the technique does not make one a lawyer.

There are other subjects (the sciences come to mind) where there is far more subject matter to master along with the technique. When studying geology you still need to learn to think like a geologist, but there is also a lot of subject matter that must be mastered. In these subjects, learning the technique is often a byproduct of learning the subject matter, or at least a byproduct of the pedagogy used in teaching the subject.

Courses are organized around parts of the subject matter rather than around technique. A well-designed program will use the technique of the field in all of the courses for that field, and will use the learning of the subject matter as an excuse to train students in the technique. Courses that try to teach only technique tend to be somewhat unsuccessful; at best they can provide a forum for students to demonstrate their technique rather than acquire it.

The academic discipline of computer science has not, I believe, done a particularly good job of recognizing the distinction between the technique and the subject matter of computer science. While there are some examples in which the technique is reasonably well described (a recent piece by Jeannette Wing[10] does a great job of describing what it is to think like a computer scientist), the seemingly non-terminating discussion of what the curriculum of a computer science major (see, for example, [1]) appears to confuse the techniques that we need to instill with the subject matter that we need to teach.

My own conclusion is that system design is really a matter of technique, a way of thinking rather than a subject that can be taught in a particular course. It might be possible to build a program that teaches system design by putting students through a series of courses that hone their system design skills as they move through the subject matter of the courses. Such a series of courses would, in effect, be a formalized version of the apprenticeship that is now the way people acquire their system design technique.

There may even be departments of computer science that have just such a series of courses. If so, I am not aware of them. They would certainly not be found by looking for schools that teach a course in system design; all of their courses would have as a subtext system design. I think it far more likely that computer science departments teach system design in much the same way that I learned system design—that there are some professors who act as master craftsmen in the field for a group of students, who apprentice with such professors by taking courses with them (often not caring about the subject matter) and learning by doing. But such training is accidental at best; often students are advised against taking too many courses from a single faculty member, which has the effect of lessening the possibility of such technique training occurring.

What would be best is a situation where an entire department was cognizant of the need to teach the design technique, and all of the courses from any of the instructors had as an admitted goal the training in such technique. Such curricula are possible in other design fields, but they are difficult to design and even more difficult to evaluate. Until we as a discipline find a way to do this kind of curricula design and evaluation, system design will continue to be learned as a craft, through an apprenticeship, and outside of the normal academic channels. Perhaps this is all that we can expect, but in times of optimism I think that we as a field could do better.

It might be that we should look not at engineering but at the studio arts for direction on such a curriculum. The approach taken there is that the students do lots of design projects, of varying levels of complexity and size, and are constantly undergoing criticism of their work, both from their peers and their instructors (and seeing the work of their peers being criticized as well). This is a lot more work, both for the students and the teachers, but seems to have some positive impact on the development of technique in an area where elegance and taste are being taught. I doubt that we could do worse than we do currently if we as a discipline were to give such an approach a try.

3.4 The Intellectual Gene Pool

Before moving on to other topics, there is one side trip that I feel must be taken while on the subject of learning system design. It has to do with what I think is an unfortunate narrowing of the intellectual gene pool in our field.

When I first started writing software, the industry was expanding so rapidly and the academic field was so new that there were far more jobs for software engineers than there were candidates with degrees in the field. As a result, lots of different backgrounds were represented in nearly every software engineering group.

For example, in the group in which I served my apprenticeship, the academic backgrounds included a Ph.D. in physics, a Ph.D. in philosophy (me), an engineer who had done graduate work in psychology, another whose background was in anthropology, and two musicians (along with two engineers who had degrees in computer science and one who had no degree at all). As a result of all of this diversity of background, there were lots of different viewpoints on any given problem, and lots of ways of looking at any task. The end result was one of the most interesting and innovative groups that I've ever been a part of.

What I find distressing is that I doubt very much if any of the members of that group who had studied something other than computer science could have gotten their first job as a software engineer today. While academia has always insisted on the proper credentials in the proper field (not surprising, given that they exist to issue such credentials), industry now requires that those who fill the job of software engineer be trained in that field. The result is that the candidates entering the profession are far more homogeneous in the way they think and the way that they approach problems. Many times they have been told what the proper way to solve a problem is, and so they simply solve it that way.

If we actually knew how to teach the way to think like a computer scientist or software engineer, and knew how to teach people to think that way, this might not be a problem. If we actually knew the answers to most of the questions that come up when producing software, getting people who already know those answers would be a way of making the industry more efficient. But, as I argued in the previous section, I don't think that we are very good at teaching how to think like a computer scientist (or at least like a system designer). Nor do I think that we have adequate solutions to many of the problems that have to do with system design in particular and software engineering in general. We can certainly get more immediate returns on our investments by hiring only those students who have a degree in computer science or a related field. But I fear that we are limiting our genetic stock of ideas prematurely, and as a result the discipline is the poorer for it.

3.5 Education and System Design

If the above observations are correct, then it is not all that surprising that system design is uncommon, and good system design even more so. Good system design requires not only talent but the training that supplies the needed technique to go along with that talent. System design is not something that can be covered in a class, but is learned through a much longer process that is more like an apprenticeship than anything else. Such apprenticeships are not the sort of thing that our educational system is set up to provide (at least at the undergraduate level), and is not going to be provided by some change in the set of courses that make up the curriculum.

In fact, most who do system design learned their craft after they completed their formal classroom education, either on the job or while doing thesis research. But changes in the economics of both research funding and the software industry have conspired against the kinds of training that lead to good system design.

4. WHERE SYSTEM DESIGN HAPPENS

If system design is in fact learned as part of an apprenticeship, there are two places that we should expect such learning to take place. The first is in graduate school, where a student can work with a single faculty member (an advisor) who acts as a master. The other is on-the-job, learning the arts of system design by doing such design.

But various forms of pressure have made this kind of training harder and harder to obtain, because less and less real design goes on either in academic research or in industry. Instead, academic research has become much more of an evolutionary task, a change that has been an unintended consequence of decisions by funding agencies designed to reduce risk. At the same time, industrial system design has become more constrained, more expensive, and less adventurous. The result of both has been not just a reduction in the ability to teach system design, but an environment in which many of the wrong things are being taught about how to accomplish that task.

4.1 Industrial System Design

Perhaps we should not be surprised that there is less opportunity to learn system design in industry, if for no other reason than that there are fewer systems that need to be designed than there were ten or twenty years ago. Industry consolidation and maturity have changed the need for system design, and therefore the opportunity for learning such design.

Twenty years ago there were far more companies creating computer systems than there are today. Further, these companies competed not merely on price but on the functionality, stability, and sophistication of the overall system, which was proprietary to the company. Every computer company had their own chips, their own hardware, their own operating system and their own programming language (indeed, IBM had three or four of each). In addition, customers buying these systems would then need custom software that went beyond the basic computer system, so there was a thriving industry in building that custom software. All of these projects required system design, so there were lots of chances to try designing a system, and lots of chances to learn either by getting it right or (often better) getting it wrong. There was also a thriving interchange of design ideas in conferences like USENIX, OOPSLA, HotOS and the like.

Current industry trends are very different. Where there used to be many computer companies, there are now far fewer. The number of operating systems has been reduced to two, with the choices being Windows or one of the Unix variants. Customers almost never purchase custom software systems, built from the ground up from specifications hammered out in discussions between the software engineers and the customers themselves. Instead, most custom software is written to allow the connection of existing systems, or the continuation of those systems on new hardware or in new environments. The production of this kind of software comes not from small companies that specialize in doing system design but rather from either the consulting services of existing companies or specialized consultancies, and is generally constrained to the

existing environments in such a way that the design freedom of the creator of the software is tightly constrained.

A lot of effort has been put into finding ways of building these custom systems in ways that are more efficient and responsive to the customer. Techniques such as extreme programming, in which small changes are made to a system with constant feedback from the customer have been developed and are widely used. These techniques emphasize doing quick prototypes and then enhancing those step-by-step until what the customer wants is produced.

Such techniques are excellent ways of making sure that the system produced is the one that the customer actually wants. But they are not good techniques if one wants to insure some form of up-front system design. Rather than trying to think out the system ahead of time by decomposing it into its constituent parts, these sorts of iterative techniques emphasize adding features by aggregation on to a first-approximation core. System design may be enhanced by refactoring as the project progresses, and there may be times when it is possible to review the entire system and change the design. But neither of these activities helps to get the project done, and often the result of such work is not visible to the customer. It is far more usual that problems in the design are coded around rather than fixed. The end result is a system in which the design emerges rather than one in which the design is thought-out.

Even worse than not being visible to the customer, work done on designing the system is not visible to the management of the company that is developing the system. Even though managers will pay lip service to the teaching of The Mythical Man Month[2], there is still the worry that engineers who aren't producing code are not doing anything useful. While there are few companies that explicitly measure productivity in lines-of-code per week, there is still pressure to produce something that can be seen. The notion that design can take weeks or months and that during that time little or no code will be written (or that which does get written will be thrown away, which often appears to be regression rather than progress) is hard to sell to managers.

The fact is that good system design takes time; it is the sort of thing that requires hard solo thinking along with long discussions with other engineers. There are days when no real progress seems to be made, and other days when the only progress is to realize that what you thought was progress over the previous few days (or weeks) was in fact a wrong turn that won't really work. Such a realization is progress (in fact, perhaps the most important progress, as it can save huge problems later in the project), but to a manager it may not seem to be moving forward.

Grady Booch once told me that he believed that the greatest contribution the tools he and others had produced to support the design process was that they made it appear to managers that the designer was doing something. He may have been exaggerating, but not by much. Anything that gives the designer time to think about the system before committing those thoughts to code helps the goal of well-designed systems.

What is really needed is an act of faith by management. The difference between someone who is making progress in coming to grips with a system and someone who is taking an in-office vacation may not be visible from the outside. Most managers are not able to do the design task themselves (those that can are rarer than those who can make the needed leap of faith), and so have to trust the system designer. Having an engineer as the designer who has been successful in the past may help a manager to be patient. But if you find a manager who is actually willing to give you time to do the design task, stick with him or her. He or she is a treasure much rarer than gold.

4.2 Design and Intellectual Property

A subtler change that has had an impact on system design is the change in the way corporations (and, to some extent, universities) view intellectual property. One of the reasons that there were conferences and mailing lists that documented and discussed system design was that the companies in which those systems were developed did not want the ideas underlying the systems to be kept secret. Indeed, the developers of the system were generally encouraged to publish their designs. Such publications were seen as ways to market the products shipped by the company, and were seen by the designers as ways of getting feedback and new ideas about the design. It also meant that there were forums where system designers could look at the work of other designers, discuss that work with them, and find solutions that could be incorporated into their own designs.

But over the past decade, the companies that funded the design work decided that they wanted to be paid when others used the results of the design. On the face of it, this is not a bad thing. If companies invested and obtained a result, it is reasonable that they be rewarded for the investment. If these companies can see that there is a reward, they are more likely to continue the investment. This is the premise behind the patent system in particular and intellectual property rights in general, so perhaps we should be surprised that there was a period when this kind of thinking was not applied to system design.

There has been much debate about whether or not software in general and system designs in particular are proper artifacts for the patent process. I'm not sure where I stand on such issues; discussions on the reification of ideas in software and the comparison of that to the reification of other inventions in a form that can be touched and manipulated, and discussions of whether software system designs are more properly covered by patent laws or copyright, are interesting as ways to fuel conversations over drinks. But like many discussions that are essentially philosophical, I'm not at all sure that they will terminate with a real conclusion.

Less debatable is the fact that the current system is not serving either the companies that fund design or the field in which the design takes place. Whether this is an inherent aspect of the system or an accident of the way in which the system has evolved is an issue that is beyond my skills to decide. But the effects are harmful in ways that I see every day.

The first problem has to do with the way that the negotiation over the value of patents occurs between the companies that hold those patents. Such negotiations, I am told by those who have been party to them, are generally done by count rather than by value. That is, company A will count up the number of patents it holds in some broad area (such as computer hardware and software). Company B will do the same. Whichever company holds the larger number of patents is the one that will be paid by the other, and the size of the payment is determined by the size of the difference. The end result is that each company cross-licenses all of their relevant patents to the other, and some amount of money changes hands.

The problem with such a scheme is that it does not take into account the quality of the ideas that have been patented. A fundamental patent is a major part of the field is no more valuable in such a

negotiation than some minor tweak that is no longer relevant because the industry has passed it by. The assumption is that, on average, any patented idea is just as valuable as any other. This is an assumption that makes such negotiations possible (since any negotiation based on the value of an idea would take forever), but it also encourages the companies involved to attempt to patent any idea, no matter how large or small, since the value of any patent is considered equal to the value of any other.

This in itself would not be a problem if the quality of patents were itself more uniform. However, the software world is still somewhat mysterious to the patent office, and was even more so when software patents first started to be issued. We can all think of patents that have been obtained for techniques that have been in common use for years, or patents for techniques that appear to most members of the profession as obvious extensions to known techniques.

I have toured the patent office, and know a number of the people who work there. They are trying hard to do the best they can, but are working with a number of handicaps. While the fees that are charged for patents are supposed to be returned to the office to fund the work that they do, in fact a considerable portion is taken and used elsewhere; the patent office is one of the few places in the U.S. government that could be considered a revenue generator. The pay that can be offered to examiners is far less than what they can make in the private law firms that deal with intellectual property law. One director in the patent office admitted to me that when examiners could only make 50% more in private industry it was still possible (because of government pensions and benefits) to attract good people, but when the differential became 100% or more it got much harder. The number of patents that are being filed has grown far faster than the number of examiners; I was told that the current wait between a filing and the time that an examiner is even assigned to a case is close to three years. Until then, applications are stored in a room filled with shelves that looks like something out of the last scene of *Raiders of the Lost Ark*.

The end result is that patents are examined in a somewhat cursory fashion by examiners whose expertise varies widely. The patent office, to its credit, has taken steps to try to make things better, but there is a 10-year history of software patents of questionable quality. Once again, this would not be a problem in itself, for the issuing of a patent does not mean that the patent is good. That, as any patent attorney will tell you, can only be decided in court when the patent is contested. But here we get to the third problem with the patent system.

Patent litigation, for those who have been through it, is the closest thing I've found to living in the world envisioned by Kafka. The theory is that a jury of ones peers can be presented with the facts of the case, and can decide if the patent in question is an embodiment of a true innovation and if the technology in question in fact infringes on the patented invention. But a jury of one's peers does not mean a jury of one's technical peers. Instead, it means a jury made up of people registered to vote in the district in which the trial is held. Indeed, having a technical background may well disqualify a person from serving on the jury in a patent case, since such a juror may be coming into the trial with a pre-conceived notion of what is novel and what is not in the field.

The result is that twelve non-technical citizens are asked to decide if something really is a novel invention, and if some other piece of technology infringes on that invention. To make this decision, the holder of the patent will introduce an expert witness, who will present his or her credentials and then testify that the invention is both novel and infringed. The defending lawyers will present their own expert witness, who will present his or her credentials and then point out how the invention in question was well known prior to the filing of the patent, embodied in a number of pre-existing technologies, and not part of the technology that is claimed to be infringed. The jury then has to decide which witness to believe. The presumption is that the patent is indeed valid (otherwise, why would the patent examiners have awarded a patent?). The end result is probably not as random as flipping a coin, but if you have gone through the proceedings it is hard to convince yourself that the results of the process actually turn on the originality of the patent and the similarity of the technology claimed to infringe on that patent.

Worse still for the subject of this work, if you have been found to infringe, there is then the question of whether or not you have infringed knowingly. If it is found that you have (rather than just infringing by accident, by re-inventing the technology contained in the patent) the damages awarded to the holder of the patent are tripled.

The impact on all of this on the discipline of system design is that companies now encourage their designers to patent any part of their design that seems novel, rather than publishing that design in a journal or talking about it at a conference. The more of this work that can be patented, the larger the patent portfolio for the company, and the less likely it is that there will be a need to pay large amounts of money to other firms when cross-licensing agreements are made. Part of patenting is that you can't talk about the item being patented until the patent is filed[4], which can be a long and involved process.

At the same time, companies are actively discouraging designers from looking at the work of their colleagues in other companies. Looking at such work can lead to future claims of knowingly infringing on a patent, which triples any damages that might be awarded. This combination of the desire to patent and the fear of knowing infringement can lead to situations that verge on the absurd. I have been asked, as part of patent filings for work that I have done, to provide exhaustive lists of any pre-existing work that might have influenced the design (known in the I.P. biz as prior art) while at the same time being warned not to actually search the literature for anything that I might not have known about previously.

While the general situation around software and systems patents is troubling, the impact that situation has had on the discipline of system design is not often acknowledged but is nonetheless large. The co-demands of keeping our own innovations secret (at least until the patent is filed) and not studying the work of others (to keep from being charged with knowing infringement) is responsible, at least in part, for stifling the discussion about systems design in the communities of software engineering and computer science. We now talk about the process of system design, or the tools that we can use to support system design, but we rarely talk about actual system designs. It is as though artists were told they could no longer talk

[4] More precisely, you can't talk about the invention before it is filed if you want to get a European patent. In the U.S., the patent must be filed within a year of the invention first being disclosed. In practice, it is hard to get approval from the legal department of a company to talk about anything patentable prior to the filing, and even after it might be difficult.

about art, but could only talk about brushes, pigments, and the way in which they prepare a canvass. It is very hard to learn about good system design unless you can see and study other system designs, both good and bad. The intellectual property atmosphere in industry has limited the number of designs that are actually talked about, and has convinced many system designers that they should not even look at the designs that are available. Whatever you think of the patent system, this effect has been bad for the overall quality of systems.

Before moving on to other topics, it should be noted that open source is often touted as one answer to the problems of the intellectual property system. Open source, it is argued, has as a major advantage that anyone can look at and study the code for a system, and hence can learn the design of that system. Good designs can be seen, as well as bad designs, and the discussion (generally on mailing lists) can take the place of the conferences that we used to have on system design.

There is a sense in which this is true, and for that I am a great proponent of open source. However, open source generally allows the discovery of system design from the artifact of the code, rather than supplying some kind of documentation that explains why the system is designed the way it is. Further, many of the well-known open source projects (such as Linux and the Apache Web Server) are implementations of existing designs. Open source projects often show us the implementation of a system design, and reading the code can teach one a lot about such implementations. But they are less useful as ways of learning about the system design itself.

4.3 Systems and Standards

The one circumstance in which most managers will allocate time for the design of a system is when that design takes place in the context of a standards body. This is also the one time that most companies will allow the designers to talk with other designers about that design. So it would seem that standards bodies would be the best place for the activity of system design. Unfortunately, for a number of reasons, standards bodies are among the worst places to do real system design.

The interaction between system design and standards bodies is complex and takes a number of different forms. At its best, standards bodies simply codify an existing technology that is so widely used that it is already a *de facto* standard. The intention is not to solve a technical problem with the standard, but to clarify and specify existing practice. This is the sort of role that the groups that standardized the C programming language or the IP protocol had. There were some technical contributions made by each of these standardization efforts, but those contributions were to clarify edge cases where the existing implementations of the *de facto* standard differed.

This is a very different role than that taken on by standards bodies that attempt to create a standard technology out of whole cloth or from an as yet unproven idea. Classic examples of such attempts are the groups that defined the Ada programming language or the OSI networking standard. The OSI networking standard gave us the seven layer model that we all know and love, but also attempted to define a standard for interconnect based on that model. Only the seven-layer model remains today. The Ada language specification defined a language that is still in use, but most of the users are required to use the language contractually, not out of free choice. In both cases, the standard was an attempt to invent and guide technology rather than codify existing technology, and in both cases the results were somewhere between partial and total failure.

One of the differentiators of standards that succeed and those that fail is where the system design takes place. If the system is designed outside of the standards process (generally by a small group or an individual) and has been implemented and used, the chances of the standard being widely accepted and useful (like the C or TCP/IP standard) are high. If the system design is done by the standards group itself, the chance of producing a coherent and useful design is much lower.

This should be no surprise. Good system design requires at least a unified vision of the overall system, and the ability to push that vision to all parts of the system. This can best be accomplished when the design is the responsibility of a single person, and can sometimes be maintained when a small group undertakes the design. However, a standards group is rarely small and unified in its vision. Indeed, the standards process is an inherently political one, where the addition of one feature is often bargained for by accepting the addition of a different feature.

This political aspect of standards groups is exaggerated by the commercial importance of standards. There was a time when technology companies differentiated themselves by the features that they were able to design and build into their systems. However, over the last decade adherence to standards has become more and more important. This is not surprising, as it allows customers of these technologies to simplify their acquisition of products. They begin with a checklist of standards, and find the vendor who can supply all of those standards at the best price. More important, by adhering to standards, a customer is not tied to a particular vendor, since essentially the same system can be bought from the competitors of that vendor.

Because of this change in the buying strategies of their customers, influence over standards groups has become very important for technology vendors. If a standard can be written in such a way as to advantage a particular vendor, the competitors of that vendor will be forced into playing catch-up for some period of time. Thus participation in and control over standards groups has become a way for technology vendors to differentiate their offerings.

The recent history of attempts to standardize various parts of the Extensible Markup Language (XML) takes this trend to something close to absurdity. In the early years of this decade, it seemed that a new standards body was being formed every month to promulgate an as-yet-undesigned XML standard. Each of these standards bodies was made up of some subset of the overall set of computer vendors, and determining which company was controlling the standards group and which was being frozen out took skills that used to be reserved for determining the meaning of which commissar was standing by which politburo member during the May Day parade.

All of this may make for good business. It may give customers more choice and more control. My only point is that it does not produce good system design. It is hard enough to do good system design when it is done by a single person or a small group whose only design considerations are technical. When that same task is attempted by large groups of people each of who has a different agenda and whose technical judgment is at least influenced by, if not subordinate to, commercial or political considerations, we should not be surprised if the resulting designs are not those that we hope others will learn to produce.

4.4 Academic System Design

If system design is best learned by apprenticeship, we could expect that system design could be learned in graduate school, where the student/advisor relationship closely models the apprentice/master craftsman relationship. This may be true for some graduate programs, but just as the changing economics of industry have made it harder and harder to teach (or do) system design in companies, changes in the economics of academic research have made it more and more difficult to do real system design there.

There is an idealized view of academic research in which that research takes greater risks than industry, plans for the longer term, and is less concerned with the commercial success of a research effort than in the intellectual content of the research. On this view, academic research can take a longer view than industrial research and development, and can take on higher-risk questions since even negative results can add to the base of knowledge that is the goal of academia. When a research program does pan out, the results can be transferred to industry for further development, and the academic researcher can turn to the next big question. Along the way, graduate students are trained in methods of research and techniques of system design, and when they are done they can either join the industrial world or return to academia to continue long-term research and the training of the next generation of graduate students.

Those who believe this will also clap for Tinkerbell.

The reality of academic research is much different than this. Professors spend much of their time writing grant proposals in an attempt to get funds for the support of graduate students. Once they get such grants, they need to target their research to produce the papers that will be accepted to the appropriate conferences and journals in their field, and be able to show the granting agencies enough progress that they will be able to get another round of grants. The cycle is actually quite short, with most grants requiring either yearly or semi-yearly reviews (and some requiring much more frequent updates). The received wisdom is that a grant needs to have enough detail to prove that the work the grant will support will in fact be successful; to do this it is in turn often necessary to have done the work already. Thus there is a tradition in some departments of using the results of the work done on one grant to get the money for the next grant. As in most systems, the hard part is bootstrapping (in this case, getting the first grant), but there is an increasingly common practice at universities to offer junior faculty seed grants for this bootstrapping mechanism.

This may not have always been the case, but the realities of funding agencies have dictated this form of risk-averse funding. The funding agencies, many of which are governmental, have been pressured to show more relevance in the research they fund, and have sometimes been embarrassed by research that has not given positive results (some of us are old enough to remember Senator William Proxmire's Golden Fleece awards, given to government-funded research projects that appeared to be meaningless or otherwise ill-advised). As the funding agencies faced more and more pressure to show that the work they were funding lead to actual results, those agencies in turn placed more emphasis on insuring that the research they funded would be successful.

One way of doing this is to require occasional "bake offs" between research projects competing for money. This funding technique uses a simple recipe. Give a number of projects seed funding for a first phase of a project. At the end of the first (fairly short) phase, have the different projects demonstrate their results. As a result of this demonstration, either re-allocate the funding favoring the most promising of the alternatives, or simply cut the funding to all but the most promising project. Repeat.

A number of government and private agencies that have been known for funding long-term research now use this model. While the model seems to make sense and certainly cuts the risk of making a major research investment in something that takes years and produces nothing but negative results, it also means that many academic research groups are in a constant short-term effort to produce the next bake-off demo.

As a result, academic research is of a shorter duration and is more risk-averse than industrial research and development. Industry is often able to invest in high-risk development based on the possibility of large returns (although this is often tied to making the results of the development into a standard, which was discussed in the last section). Academics are increasing unable to convince granting agencies to fund for the same long duration.

Nor are academic institutions much more open to sharing the results of their research than is industry. The lesson of intellectual property has not been lost on many of these institutions that now seem to hope that the developments of their research can be used to add to the endowment of the university. I do considerable collaborative research with various academic institutions, and have noticed over the past five or so years an increase in the difficulty of negotiating agreements on the intellectual property generated by such collaborations. Indeed, one collaboration that I tried to fund a couple of years ago became impossible when the academic institution's lawyers insisted on terms that gave the institution all rights to anything that was done by anyone in the collaboration (including any work done entirely by my group inside of Sun). Even when the conditions are not so irrational, the desire by these institutions to patent the result of the work of their faculty and graduate students has had the same squelching of open discussion as has been caused by the protection of intellectual property in industry.

Whether such policies will lead to more money for universities is yet to be seen, but these changes in funding and sharing do mean that it is less likely that full system design will occur at these academic institutions. Academia is subject to the same pressures as industry (although the pressures may come from slightly different sources), with the same results with respect to system design.

5. WHAT DOES IT ALL MEAN?

The previous sections paint a rather grim picture concerning the practice of system design. A combination of impatience, economic pressures, and a lack of trust by those who don't understand what is required for system design seem to be creating a perfect storm, where we don't have the time or support to do real design in either academia or industry, and where we can't train the next generation of system designers in the craft.

Perhaps this is just a sign of the age of the author, and all of the trends that I have identified are simply changes that have made the world different and to which I should simply adapt. I could be convinced of this if I didn't see a real desire in the next generation of engineers and computer scientists to learn something about system design. It isn't that they have gotten beyond the need to design systems; when they see a good system design they are appreciative, excited, and want to know how to create designs that have the same quality. They may not be able to verbalize what they

are missing, but they know it when they see it, and they would like to learn.

Another possibility is that the lack of system design at this time is just part of a natural cycle of development in the field of computer science. On this view, we are in the analogue of what Thomas Kuhn[5] called a period of normal science, in which the existing theory (or system designs) were being confirmed, tested, and slightly altered. Perhaps the systems that we have are good enough for what we need to do, so there is little or no need to do major design work on new systems. That will change in the future when we find tasks for which the current systems are inadequate, but until we do we should expect little support for system design. Indeed, systems like those being developed by Google are just the kind of radical departures that we would expect in a time of revolution, and they are indicators that we are about to enter into a new system design cycle.

I have some sympathy for this view, in that it gives me hope that things will change. But I also realize that this view is based on the false assumption that there are fewer systems being produced now than there were in the past. In fact, I observe all kinds of systems being produced, from the service-oriented architectures of web services to the ontologies of the semantic web. What I find missing in these systems is a notion of design other than the designs that are done in standards committees or other large groups, or designs that emerge from the code that is thrown together to implement the system.

I think one explanation can be seen if we re-read Ivan Sutherland's *Technology and Courage*[9]. System design, like any other form of research, is hard work that entails taking great risks and therefore requires the constant application of what can only be called courage. It takes courage for an engineer to design a system without constantly asking the customer if it is what the customer wants. It takes courage for a manager to trust an engineer to take the time to design a system. It takes courage for a funding agency to underwrite an academic research project that might well fail. It takes courage for a company to back a design that has not been blessed by a standards body. What we are lacking today in our industry is the courage that is needed to take the kinds of risks that are inherent in doing system design. Whether this lack is caused by the scarcity of funding, or the bursting of the technology stock bubble, or the consolidation of the industry is hard to tell. But the reason that we are no longer designing interesting systems is, I believe, simply a lack of the courage needed to do so.

If this is true, then one possible approach would be to solve this problem ourselves, at both the individually and collective level, by simply insisting that we be given the time and resources to do good system design. Finding courage is difficult, and instilling it in others more difficult still. But either is less difficult than changing the economy, or the legal system, or the attitude of the funding agencies, or the ways in which our field is taught. Indeed, we could make the change starting with ourselves, by taking the time and making the effort to do good system design, and to demand of our colleagues (and managers) that they both give us the opportunity to do such design and do such designs themselves.

But given the realities of our industry and the wider economy, I hold little hope that simply making such demands will solve the problem. But this doesn't mean that the situation is hopeless. Instead, it means that those who wish to continue in the craft of system design need to find other, less direct, ways of allowing such design to be practiced and taught.

I am actually encouraged by some signs that this is already happening, although perhaps not in the way (or in the places) that any of us might have expected. These signs are not coming from industry, where the relative power of the engineer and the manager has changed to the advantage of the latter, and where managers are under increasing pressure to cut costs and therefore have become more and more cautious. Nor do I see much change in academia, where short funding cycles and publications by the pound are still driving out good system design. Where I see encouraging signs are in two areas that are generally not thought of as central to system design, the areas of agile methods and open source software.

"Agile methods" mean lots of different things to lots of different people, so I should begin by saying what I take them to be. This is not because I think that my characterization is any better than any of the others, but simply because it will help in the discussion that follows. Like patterns or open source (a discussion that will follow) here is considerable theology in the characterizations of agile methods, and I don't wish to get caught up in such theological debates. I'm happy to admit that my characterization is not really what is meant by agile methods; what I am describing is a trend I have seen in development that is at least sometimes given that label.

What I am using the term "agile methods" to label is an approach to writing code (and, ultimately, systems) that is based on small groups of programmers working closely together; in the most extreme form of this the small group is a pair of programmers working together with a single keyboard and screen. No matter what the size of the group, the system is built by iteratively constructing small pieces, and then enhancing that working system in small, manageable chunks to build the ultimate large and complex system. In addition, I include the practice of "test driven development" in which the tests for some piece of functionality are written before the code that provides that functionality. There are, of course, many other techniques that get included under the term "agile methods," but for the purposes of this discussion these are the features that are most important.

Earlier I noted that such an approach to the production of a system seems to be an invitation to plunge into the code before thinking things through and then to make incremental changes to the undersigned system until things are good enough. Such an approach seems to actively discourage thoughtful system design. And, indeed, I have sometimes seen these methods produce systems that were badly designed, overly complex, and not well thought out. What has surprised me is the number of well-thought-out systems whose designs show taste and elegance that have been produced using these techniques.

The reason, I believe, has to do with two of the aspects of such agile methods. The first is the combination of breaking the overall system down into small pieces and the requirements of test-driven development. Each of these techniques requires that some thought be given to the abstractions that form the system. Breaking the system down into smaller pieces requires some thought into what those pieces are going to be and how they fit together, which is exactly the art of system design. In order to write the tests before the code that is to be tested, an abstract notion of what the code is supposed to do must be thought through. In deciding what to test, a programmer needs to think about the general functionality of the system, and how that functionality is going to be accessed. Both

activities require thinking about the interfaces for the various components of the system in a fashion that is one removed from the implementation of those interfaces. By deciding what small thing can be done and by writing the tests first, agile methods impose a requirement of thinking about the abstract system that is a way of expressing the overall design of the system.

The second, and more important aspect that favors system design when using agile methods is that those methods require that the work be done in small groups, each member of which needs to understand the entire artifact. This in turn encourages discussion of the overall system, not just as the level of the code that is being produced but at the level of the system itself. Each member of such a team has to explain to the others how the system fits together, and just that act of explanation requires thinking about the design. Even better, the others can then help to make the overall design better; the give-and-take of a small-group programming session is much the same as that found in a good design session because it is, in fact, a design session.

What is important here is the required communication between the participants. Having to express a design will often uncover problems with the design, and can certainly show areas where the design (and, therefore, the communication of the design) is unclear or inconsistent. While it is true that writing down the design of a system is a form of documentation that can help people who want to learn or understand the system, the greatest benefit of such a written design is to the designer who must do the writing. The very act of writing the design document helps to clarify the design itself. In the same way, having to communicate the design during group programming helps to clarify and simplify the design.

The process of small group development also provides a opportunity for the members of the group to serve their design apprenticeship. While the group may not consist of an acknowledged master and a set of apprentices (although it could), the constant discussion of the design even with a peer group can help in the development of taste and craftsmanship. While there is always the possibility that bad taste will be reinforced and bad habits encouraged, the process of peer-mentoring is better than no form of design feedback at all.

Whether it be to a group of peers or a master, the real point is that the design needs to be expressed to someone else. It is very difficult to mask the weaknesses of a design when you are communicating that design to someone else who is intimately involved in the implementation of the design. Simple designs can be communicated easily; complex designs are hard to explain. Just as writing down a design will often show flaws or weaknesses in the design, explaining a design to a peer will often improve the design.

Working on an open-source project also provides engineers both a forum for the discussion of design and a mechanism for learning through an apprenticeship. The first of these is supplied by the mailing lists that are central to many open source projects. On these lists there is constant discussion of the design alternatives, philosophies, and trade-offs that are faced by the overall project. Newer or less experienced engineers can ask questions that will be answered (and discussed) by the overall community. Like the discussion that goes on between the members of a pair-programming team, such electronic discussions allow the engineers to try out ideas, have those ideas criticized or amplified, and generally participate in the design process of a large project. The discussions tend to be at a different time-scale than those held face-to-face with a pair-programming partner, and often involve a much larger group of participants. But they are still forums that require discussion of the design. Better still, they are forums that require that the participants communicate the design in a clear and persuasive way. Just as the act of communication between two programmers can help to clarify and simplify the design of a system, the act of communicating a design to the other members of an open source project will help to clarify and simplify the design of the open source system.

These discussions often replicate, at least electronically, the master/apprentice relationship that is so central to becoming an accomplished designer. Such relationships are established in spite of the mythology that has grown up around the way open source projects are run. The establishment of this sort of mentoring happens because of the reality of the way that open source projects work, a reality that is very different from the folk wisdom that has grown up around such projects.

The folk wisdom of open source, best exemplified by the writings of Eric Raymond [7], holds that open source projects are chaotic, highly democratic undertakings in which the marketplace of ideas sorts out the good ideas from the bad, the code is written by anyone, and there is no hierarchy. In actual fact, most of the successful open source projects are run as semi-benign dictatorships in which a very small group of people controls all of the code that is put into the project. These people are the committers of the project, and no code is allowed into the source repository until it meets their standards.

It is true that anyone can offer code to the committers to see if it can be included into the project. But most of the code will go through a very detailed reading by the committers, and only be accepted when it is found to be good by the standards set by this group. Not surprisingly, most of these committers are just the kinds of master craftsmen of code that you would want supervising the apprenticeship of those learning system design. The apprenticeship is not as direct, with little or no face-to-face discussion, but the overall process is the same. The apprentice will try to solve problems, offer his or her solution, and be told to try again (generally with some discussion as to the reasons for needing to try again) until the code and the design is right. The communication may be electronic rather than face-to-face, but the process is the same as it was 20 years ago; one of trial-and-error, of frustration and trying again, and of failure and (hopefully) enlightenment (or at least increased mastery).

This is a process that benefits both the apprentice and the master. The apprentice benefits in obvious ways, learning how to be a better craftsman and gaining a better understanding of how to build and design a system. The master benefits by using the apprentice as an idea magnifier. By having others doing some of the work, the master is freed to concentrate on those parts of the design or the code that only he or she can do. The end result is that the kinds of systems that can be built are more significant, and the ways of approaching design are conveyed.

This ability to learn, to teach, and to tackle hard technical problems without the oversight or interference of management is also, I believe, one of the prime reasons for the popularity of open source projects among engineers. Such projects are places where technical decisions can be made on technical grounds, and where the decision making powers are given to those who have shown technical ability in the past. The fact that the end result of such developments is innovative software that is often superior to that produced by the

projects that are the day jobs of the very people who write the open source software may be ironic, but it should not be surprising.

In an important sense, both agile methods and open source can be seen as reactions to the difficulty of doing system design in either the academic or the industrial world. One solution to this could have been confronting the managers, professors, and funding agencies that have made it increasingly more difficult to do system design in the traditional environments. But this other solution is both more indirect and, in many ways, more in keeping with the ethos of software design. Rather than trying to change the set of constraints that frame the problem, designers and those who wish to learn design have simply designed around the problem. By adopting agile methods, we have found a mechanism that allows us to discuss and learn design without having to tell our management that this is what we are doing. By working in open source, we have created an environment in which we can continue to do technical work framed in purely technical way. The fact that open source needs to be done on our own time is a minor inconvenience; most good software designers would prefer doing technical work to most other forms of recreation. In a meta-sense, the new venues for learning and teaching system design are themselves excellent examples of system design, in which a problem is solved in a fashion that is elegant, subtle, and pleases both the practitioner of the art and the consumer of the code. The end result is that the craft survives, thrives, and continues to evolve.

6. ACKNOWLEDGMENTS

I would like to thank Bob Sproull, Ivan Sutherland, Margo Seltzer and Ann Wollrath, all of whom have been generous with their time and ideas during discussions of much that is contained in this paper. Special thanks go to Brian Marick, whose care and comments during the shepherding of this paper have greatly improved the result.

7. REFERENCES

[1] *ACM Curricula Recommendations***,** http://www.acm.org/education/curricula.html, 2005.

[2] Brooks, F.P., **The Mythical Man Month: Essays in Software Engineering, 20th Anniversary Edition**, Addison-Wesley, Boston, MA, 1995

[3] Brooks, F.P., *The Design of Design*, Turing Award Lecture, http://terra.cs.nps.navy.mil/DistanceEducation/online.siggraph.org/2001/SpecialSessions/2000TuringLecture-DesignOfDesign/session.html, 2000

[4] Hoffman, Daniel M. and David M. Weiss (ed), **Software Fundamentals: Collected Papers by David L. Parnas**, Addison-Wesley, Boston, MA, 2001.

[5] Kuhn, Thomas, The Structure of Scientific Revolutions, University of Chicago Press, Chicago, IL, 1962.

[6] Lampson, Butler, *Hints for Computer System Design*. ACM Operating **Systems** Rev. 15, 5 (Oct. 1983), pp 33-48

[7] Raymond, Eric, **The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary**, O'Reilly Media (2001).

[8] Ryle, Glibert **The Concept of Mind**, University of Chicago Press, Chicago, IL, 1949.

[9] Sutherland, Ivan, *Technology and Courage*, Sun Microsystems Laboratories Essay Series, Mt. View, CA, 1996

[10] Wing, Jeannette M., *Computational Thinking*, Communications of the ACM, Vol. 49, Issue 2, March, 2006.