Abstract. Why has a problem between “systems engineers” and “software engineers” been documented every year of INCOSE’s existence? Is it because there is a difference between systems engineering of hardware-intensive and systems engineering of software-intensive systems, and this difference is not understood or respected? This paper presents several of the issues involved and presents opposing factors for consideration by those who will have to face them, such as hardware systems engineers addressing systems with more and more software components.

The author posted fourteen assertions, based on personal experience, to the INCOSE discussion list server. Twenty responses were received, providing views, examples, and additional information. Respondents agreed on a number of differences between the systems engineering currently performed on software systems and that on hardware systems. Those differences are thought to be decreasing as software evolves as a field. Recommendations are made for changes that may improve both dialogue and system development.

Open questions generated by the discussions are suggested in the hopes that research efforts may address questions of importance to the improvement of the engineering of systems.

INTRODUCTION

Increasingly, software is appearing as a component in systems that used to be strictly hardware. (Armstrong 1997), (Rechtin 93) As systems engineers who have learned the trade in hardware systems begin working on systems with a significant software component, problems appear. Over twenty-five papers have appeared in INCOSE and related proceedings discussing aspects of “the systems-software problem.” Addressing topics ranging from relationships (Downward 1994), (Swift 1993) and training (Harbaugh 1993) to integration of processes and tools (Alford 1992), (Rose 1997), these papers collectively define a chasm of misunderstanding between the two areas.

In addition to the clash between cultures and methods of software engineers and systems engineers, there also seem to be misunderstandings among systems engineers: between those who have worked primarily on software systems (usually having started as programmers) and those who have worked primarily on hardware systems (usually having started as mechanical or electrical engineers). INCOSE members still seem to fall primarily into one category or the other: few, as yet, speak both languages fluently.

To clarify whether these misunderstandings are due to something inherent in the nature of the systems that were being engineered, the author posted on the INCOSE list server a set of fourteen provocative assertions about how systems engineering differs when engineering hardware systems or software systems. The post generated twenty substantial responses. In the responses, differences were confirmed, but also many points of commonality were found. Indeed, there seem to be trends towards additional commonality, which are discussed below. Differences were found primarily in the following areas:

- Many hardware requirements derive directly from physics, and are not generally called out in specifications directly. (Point 1)
- Software-intensive systems are going farther into the user domain than hardware-only systems ever could. (Point 3)
- “Hardware” is less cohesive a term than “software.” (Points 3 and 4)
- Requirements change more for software systems. (Points 1 and 6)
- Software engineering is a younger field than hardware engineering, and software systems engineers are correspondingly younger than hardware systems engineers. (Points 6 and 9)
- Software complexity issues make it important to add independent testers in the system test phase. (Point 2)
• There is a lack of confidence that software systems engineers and developers are trained enough in engineering or pay enough attention to the “ilities.” (Points 7 and 8)

These and other ideas are analyzed in the context of “points” and “counterpoints,” in the hopes of stimulating further discussion and understanding.

**METHOD**

From the twenty list server responses, quotes were selected for the paper. “Counterpoints” from the respondents, both those that related specific assertions (“points”) and general counterpoints, were analyzed. Respondents were quoted unevenly, with quotes in this paper primarily coming from two kinds of respondents: those providing significant insight into differences or providing real-life examples, and those speaking with authority in the software and software systems engineering fields. The latter were included to balance the author’s primarily hardware systems engineering background.

The premise of the initial list server post (the question) is given first below. Then comments related to the premise as a whole are summarized and analyzed. Following are some of the individual points made by the author in the initial post, and a summary of the counterpoints to each one. Finally, a general counterpoint is discussed, along with open questions raised by the discussion. The hope is that research into these open questions will improve the engineering of systems, both software-intensive and otherwise.

**QUESTION**

In what ways does the systems engineering of primarily-hardware systems differ from the systems engineering of primarily-software systems, and why?

**OVERALL COMMENTS**

This question clearly struck a nerve. There was probably more list server discussion on this than on any other technical issue within the past year. However, only a handful of respondents claimed experience engineering both software-intensive and hardware-intensive systems—most systems engineers still consider themselves primarily one or the other.

One point that many respondents made eloquently is that there is more commonality than was implied in the original post. Software systems engineering can be very similar to systems engineering of hardware systems, provided those systems are mostly unprecedented and have high problem-space complexity. These two characteristics of software systems seem to be generally agreed upon. More description follows under the various points.

In addition, commonalities are increasing. Many of the differences that do remain relate to the newness of the software field. This is reflected in the increasing numbers of components and packages available for reuse. It is also reflected in the changing nature of software development processes, including software systems engineering. Many respondents spoke of the software development field as being in its infancy compared to hardware development, which has its roots in the industrial revolution of the nineteenth century. Odious documentation processes were a favorite topic in this vein.

**CRITICALITY OF REQUIREMENTS**

Point 1.

Proper specification of requirements seems more critical for systems engineers on a software system. Software can do anything logically possible. If there are no requirements, there is no software. Hardware systems, on the other hand, are constrained by material properties and the laws of physics…. You only need to have requirements for the things that make you pick one element over another. And defining in great detail the allowable end-of-life degradation of battery cells can be a waste of time after the cells have been chosen, unless there is a chance of changing the cells. Hardware does what it does.

Summary of Counterpoints. “Requirements are critical for unprecedented systems whether they will consist of software, hardware, or both. The purpose of requirements is to decide what characteristics of the system will satisfy the customer.” (Gurwell 1997) “How the requirements are implemented guides what to buy.” (Kasser 1997)

The apparently greater importance of requirements to software systems is due in part to the fact that many are unprecedented, and to the paucity of ready-made objects for reuse in many domains. “For instance, it’s a relatively simple matter to specify that I need a ‘spread-spectrum multiband VHF telemetry antenna.’ … In the software case, the software engineers often

---

3 The terms, “software systems,” “primarily software systems,” and “software-intensive systems” are used interchangeably here, as are the comparable terms for hardware systems.

4 All points are from the initial list server post by the author. Some are edited for clarity. Four of the original points, those that generated little enlightenment, are not addressed in this paper.

5 As constrained by the semantics of the chosen programming language, of course.
Some software projects write detailed requirements that do not just capture black box-type functional requirements, but instead achieve a level of detail better considered software design. In this case, system test ends up testing software structure rather than just satisfaction of customer needs. (Lister 1997) The test phase may be used to fix things: “In response to the question of different failure modes between hardware and software, ‘play with hardware long enough and it will break, play with software long enough and it will work!’” (Hyman 1997)

Regarding whether hardware systems engineers can try to break things, (Wittig 1997) noted that “we will use up almost 200 steering systems for testing of a single steering system design for a single customer vehicle model.” (Owens 1997) noted that it depends on the industry whether hardware systems are intentionally broken. In the electrical utilities field, “I…test in the worst of circumstances for as long as it takes for a component to fail…. [In aerospace,] we still break hardware (sometimes in quantity), but it is not our goal. The difference comes in part because of the long lead times for replacement parts…and the cost impact of space hardware.”

WHO TESTS THE SYSTEM?

Point 2. Hardware system testing is usually done in concert with the hardware system designers, but in software system development, a generally accepted truth is that a separate group should test the software.

Summary of Counterpoints. All designers, both software and hardware, test their work before system test. For reasons that vary from immature development practices to “the need to make sure things that were supposed to be done were done” (Kasser 1997), independent and objective system evaluation and test help improve the quality of software products. But “software tests can’t be run without any assistance from the domain experts and developers. They are needed, at least, to isolate the errors.” (van Gaasbeek 1997)

Unit testing must be performed on both hardware and software systems prior to system test. Both need the designers present for guidance ad troubleshooting during system test. Nevertheless, the software complexity described above under Point 3 makes it valuable to engage additional minds on the final verification and validation of software and software systems.

Software system failures are harder to detect during testing because “the state-space for software is so large, and that the state-to-state transitions can be so complex.” (Tockey 1997) “Unlike most hardware experience, software can go a long way before a critical or devastating flaw is found.” (Hongsermeier 1997)
Specialties within software engineering (and software systems engineering by implication) are helping to tame complexity, which software systems often develop much more rapidly than well-designed hardware systems do. This is partly because software is all in the logical realm: interference and side effects among various parts of software can be very difficult to predict. In hardware systems, it is the exception rather than the norm when two types of interfaces collide. One simply assumes electrical signals will not interfere with an antenna deployment and electrical power surges will not cause spin balance problems.

Some respondents answered that all systems, especially unprecedented systems, need an architect, although these roles might be called the Chief Engineer or other title. It is true that all good systems need a small number of architects, often one. (Rechtin 1991) With the proliferation of software specialties and the increasing availability of levels of software packages, perhaps software complexity will be easier to manage.

Another issue is the purpose of different systems. (Gabb 1997) says, “Software intensive systems are going further into the user domain than hardware only systems ever could, and the complexity of requirements is to be expected.”

(Tockey 1997) agrees: “The kinds of things generally solved by pure-hardware solutions tend to be relatively simple in their problem-space perspective, but can be quite complex in their solution-space perspective…. Software systems tend to be extremely complex in their problem-space perspective and (clearly with exceptions) simpler in their solution-space perspective.

“For example, I think we all understand what a radio is in principle; the real trick is how to make one that satisfies the customer’s needs for size, weight, clarity of reception, battery life, etc. while still meeting our need for manufacturability and profitability. In contrast, if you could precisely define what is meant by ‘flight management,’ then I propose that most of the complexity in building flight management software has already been licked.”

Software complexity is logical, and often the province of requirements (the problem side) as well as than design (the solution side). Hardware complexity is physical, and generally has been tamed, to some extent, by a long history of developing objects with simple interfaces, like RF receivers and transmitters. Perhaps when developers get better at encapsulating complexity within software objects, software requirements will look more like hardware requirements.

Finally, hardware complexity is increasingly being simplified by allocating it over to software. At Dulles airport, the mechanical lockers that used to operate by means of quarters dropping through a slot now are run by a computer that takes Visa, asks for the number of a locker, and gives you a retrieval password. No more quarter slots! Much simpler mechanically!

**SCOPE OF “SYSTEMS ENGINEERING WORK”**

**Point 4.** In my experience, hardware systems engineering tended to include any job that no one else wanted to do (“Miscellaneous” work). Although I had never studied passive intermodulation products in communication systems, and their exacerbation by loose metal-to-metal contacts, (my first response was, in fact, “Huh?”) I was nevertheless directed to resolve this system problem. Of course I had to bring in experts -- that was part of my job. This role seems to be less common for systems engineers of software systems, who often have a more bounded role.

**Summary of Counterpoints.** Limitations on the scope of software systems engineering were asserted by respondents (both those with software and those with hardware backgrounds) to be due to a lack of basic engineering background (see Point 7) or a narrower sense of what is their domain.

(Honour 1997) objected to people considering “hardware” to be a “thing” comparable to software, when all it really is is a collective name for “anything other than software.” This would explain why the complexity of hardware is better apportioned out among many experts than software complexity is:

“Maybe what you are seeing is that software systems engineers are working in a narrower domain, and can be expert in that domain.” (van Gaasbeek 1997)

Where software systems engineers are assumed to operate in a software-only domain, this tends to flow from the culture of the organization and what it defines as the role of systems engineers. (Requirements Owner is a particularly popular software systems engineering role, and Coordinator of Disciplines relatively unpopular.) (Sheard 1996) If the culture allows, good systems engineers for any type of system will do whatever needs to be done, calling in experts as needed.

**NATURE AND OWNERSHIP OF INTERFACES**

**Point 5.** Hardware systems engineers have to “own” internal interfaces. If simply relocating battery cells within a battery pack can save millions of dollars of launch costs due to reduced balance weight require-
ments, then the systems engineers are the ones who break the unpleasant news to the battery people. In contrast, software element designers (rather than the systems engineers) must own internal interfaces, because software interfaces must be built in to each software “unit.” Early meetings between the two interfacing groups, perhaps with a systems engineer in attendance, must establish an agreement as to the contents and format of the interface, after which the agreement must be incorporated fully into both sides.

Summary of Counterpoints. Historically, interfaces have been encoded directly into software elements. This, however, ensures those elements are not easily reused. As software becomes more object-oriented and commercial components become widely available, “just like the hardware engineer, the software engineer will have to examine the ’pins and sockets’ of the software module and understand the properties of that module if he/she wants to use it…. The real challenge for the development of software module catalogues is that of defining an industry standard interface - the equivalent of the ’dual in line’ IC packages and TTL/CMOS interface tolerances.” (Lister 1997)

Like requirements issues, the apparent difference between hardware and software in interface management is likely due to the abundance of available hardware COTS objects and the relative scarcity of comparable COTS software objects. As software becomes more object-oriented and reusable, this distinction should decrease.

BACKGROUND AND EMPHASIS

Point 6. “Systems engineers are older, more cynical, and harder to educate than software engineers.” (Brook 1996)

Most hardware systems engineers were educated prior to the days when razors included 17 kilobytes of code. Thus “software” can seem somewhat mysterious and troublesome to them. “If software is just a subsystem, why is it so late? And if it is just writing programs, why do they need final requirements before the hardware people do, who have long-lead items to buy? And why are they so difficult when the customer needs change and we need to tweak requirements?”

In contrast, software systems engineers have seen that continuously changing requirements can not only be costly to manage, they can absolutely kill programs with nothing to show for the money spent. Because of this, they may place too little emphasis on the customer interface role. “We have the requirements, now we have to develop the system to meet them, and we can only do that if they leave us alone!”

Summary of Counterpoints. The Brook quote certainly got the attention of systems engineers -- jokes about age abounded, and one respondent countered, “A ‘cynic’ is an optimist’s word for a realist.” (van Gaasbeek 1997) Yet no one highlighted the difference (or perhaps it best described as a lack of difference!) between software engineers and software systems engineers.

One set of counterpoints emphasized the criticality of managing the customer interface role well. The customer is not always aware of how important his early participation is, to solidify what is really needed, but if the customer gets too involved, requirements can undergo too much change (“creeping featurism”) (Schroeder 1997), hindering the development of anything real.

“The difference between a new software requirement and a new hardware requirement is that the latter is physically tangible for the most part, thus the potential impact can be easily understood. Decision-makers who are not well schooled in software may not recognize the true impact of a software requirements change, since it is usually abstract. This should improve with time, which in turn will improve the creation of respectable schedules that include software.” (Schroeder 1997)

There is agreement that both software systems engineers and software managers tend to be younger than their hardware counterparts, which many attributed to the relative newness of the software field. There is opportunity for improvement in cross training. The situation is improving: more systems engineers have software backgrounds than ever before, and more software engineers are broadening out to systems engineering.

ENGINEERING DEGREES

Point 7. Hardware systems engineers often have engineering degrees. According to a colleague, this means they are made aware in college of their responsibility to produce safe products for society. In contrast, software systems engineers tend to have mathematics or computer science degrees. This means their training tends more to the logical than to the societal.

Summary of Counterpoints. Software engineering (and software systems engineering) as disciplines are very new, and there are disagreements on what they mean, even among practitioners. “My point is that

6 Some people think software engineers eliminate the need for systems engineers or systems engineering. One practitioner at a systems engineering appraisal even asked, “Why am I here? My system doesn’t have any hardware; we don’t do any systems engineering!”
software is not presented as an engineering discipline, not that it is not capable of being an engineering discipline." (Tockey 1997) “Regrettably, it seems to be widely true...that most specifically degreed 'software engineers' have never received the classical engineering training.... Software engineers are often not thoroughly acquainted with basic physics, statics, dynamics, chemistry, thermodynamics, physical chemistry, or any of the standard subjects found in an engineering curriculum.” (Hongsermeier 1997)

More than one respondent noted that there is a difference between a computer scientist and a software engineer, and there are fewer of the latter.

Regarding engineering responsibility, “the safety features that software does commonly control are new and exciting. For example, new command and control systems have extensive features to assure commanders and GIs that stray bullets will not come their way. In addition, the discussions and debates regarding the social responsibilities of the software engineers regarding individual privacy are numerous, sweeping, and (remarkably) sometimes very subtle in their scope and implementation.” (Hongsermeier 1997)

ADDRESSING "ILIITIES"

Point 8. Hardware systems engineers must be familiar with more of the “ilities” than software systems engineers must. Software does not go bad with time, nor does vibration mess up its algorithms, to paraphrase another colleague.

Summary of Counterpoints. Software does need to be concerned with understandability, verifiability, modifiability, reusability, maintainability, portability, reliability, predictability, flexibility, and usability. “Every time I fly in an aircraft that has a Full-Authority Digital Engine Control System, or a digital flight control system (without a hardware backup), I say a little prayer that the software engineers did, in fact, consider the safety "ility." (van Gaasbeek 1997)

Some respondents countered that software does indeed go bad with time, either due to obsolescence, or due to maintenance, which in software is really continuing modification rather than greasing or replacing worn parts. "Hardware gets better when it is maintained, software gets worse." (Tockey 1997) A good computer-system development effort will ensure the architecture allows for timely replacement of obsolete software and computer system hardware.

One respondent noted that while algorithms are not messed up by vibration, software is increasingly being asked to control physical characteristics of systems, including the environment in which the software runs.

In general, there is a lack of confidence that software developers pay enough attention to the “ilities.” This is an opportunity for improvement of the field, and increasing the emphasis on engineering of all kinds within software development teams would help.

PROCESS FOCUS

Point 9. Software systems engineers seem better attuned to process than hardware systems engineers. Software development has more mature processes than systems engineering does, because certain repeatable actions are always included when building software. Hardware processes have been mature for a long time: some as electrical or building codes, for example. But in the “Miscellaneous” arena where hardware systems engineers seem to operate, there are few, if any, repeatable processes. If I’m taking a job because no one else owns it, I have to invent my own process.

Summary of Counterpoints. While it is true that documented software processes have received a lot of attention in the last ten years, many respondents concluded the following:

1) a small minority are actually following documented processes, “In my opinion, code-sling first then backfill the 2167A documentation is a popular (software) approach” (Tockey 1997) and
2) the newness of software engineering compared to hardware engineering has led to a need for more documented processes.

Unlike the documentation of the software itself, which is almost always important, documentation of the processes can be burdensome and non-value-added, and where it is, it gives all process formalization a bad reputation.

Hardware engineering (electrical, mechanical, etc.) has developed many standard processes over the years that are widely agreed upon and taught in all degree programs. Hardware processes are likely to be embedded deeper within the organization and individual engineers. Thus they are less documented, less obvious, and perhaps less easy to improve. Certainly there is no widespread “Hardware CMM” effort comparable to the SEI’s CMM® for Software.

Finally, while any specific “miscellaneous” task has not been done before, the analytical work done in solving the problem must have a repeatable basis. In fact, the scientific method depends on a basic principle of repeatability. In the example in Point 4, I first found out as much as I could about the problem by inter-

7 Capability Maturity Model is a service mark of Carnegie Mellon University. CMM® is registered in the US Patent and Trademark Office.
viewing experts and reading references. I then listed several possible ways to fix the problem. These were analyzed and evaluated against a set of criteria that included easy implementation (cost, schedule, and producibility), and being a permanent fix (quality). The trade study was reviewed (validated) with appropriate people, and a plan was developed to implement the winning solution. This very repeatable process has been defined in many systems engineering texts and was well known and commonly used by my satellite systems engineering colleagues.

**OBJECT ORIENTATION**

**Point 10.** Hardware is and always has been object-oriented. Systems engineering texts from the 1960’s discuss objects and attributes using much of the language which now brings software modeling methods to mind. “Software reuse” is growing, but is not yet producing anything like the 1000-page catalogs of electronics parts I leafed through in the early 1980s. As a result, systems engineers for hardware systems may not even know the difference between functional and object-oriented design. All their designs are object-oriented, and in fact, it takes a true system engineering approach to first address “What” needs to be done (the functions) rather than jumping straight to “How” it can be done (with what components, designs, etc.)

Software systems engineers, on the other hand, see a great deal of difference between one-time-only software designed strictly functionally, and a completely different approach using objects with encapsulated functions. Still, none of the OO texts I have read clarified well for me how software system functions can easily be turned into software objects -- that still seems to be an art.

**Summary of Counterpoints.** Respondents agreed that system function needs to be turned into objects both in hardware systems and in well-defined software systems. Respondents agreed that turning functions into objects is an art for both software and hardware systems: “The impulse to jump directly to objects is a great obstacle to successful solution discovery in society in general, not just in technical fields. Thinking functionally is a very slippery skill.” (Cutler 1997)

This last respondent also noted that while reuse is still something of a dream for one-time software development programs, there is considerable reuse in the commercial world: “The Microsoft Windows developer application toolkits are a massive specification and part catalog. Consider that every one of your X-Windows or Windows 95 or Macintosh application programs looks like every other one, and that you can usually run any application straight out of the box with little or no preliminary introduction. That’s a direct result of object reuse, and the software designers select the components out of a catalog, just like a EE or structural steel engineer does.”

However, a caution has been noted. All systems engineering capability models (see Sheard 97) emphasize the need to investigate various design options, and trade them off against requirements, when selecting the system architecture and design. Yet some object-oriented methods tend to begin with object-oriented requirements and gradually develop them into designs without ever explicitly trading off candidate architectures or designs. (Lykins 97) Would these designs be improved by adding such a step?

**GENERAL COUNTERPOINT**

Several respondents separately insisted that systems engineering is the same whether the system being engineered is primarily software or primarily hardware. Very few systems any more are just hardware, and no software system stands alone without at least computer hardware. Further, increasing amounts of software are going into what used to be all- or mostly-hardware systems. The following are some examples of comments:

“I have worked on projects with hardware and imbedded software, hardware and real-time software, and software intensive systems… In all of these systems I was taught by experienced engineers, many now retired, how to design, develop, implement, test, produce, and maintain a product using a systems engineering approach. I have learned is there is no difference in the systems engineering process when applied to any product, except in scope or the tasks at a design level that focus on a particular discipline.” (Nallon 1997)

“We've been getting emphasis on incorporating AIS/IT acquisition practices into our DSMC curriculum… I was very hard-pressed to find any DoD system which isn't AIS, IT, C4ISR, or software-embedded systems...In fact, the only current system I find which may be around in a few years without software is the M-16 rifle. Bottom line: The success of almost all future technology is heavily dependent on successful integration of hardware and software, and yet we still treat software separately.” (Zittel 1997)

**OPEN QUESTIONS**

The list server debate raised many interesting questions, only a few of which could be addressed above. The following additional questions would make good research projects, and are offered in hopes of stimulating useful discussion.
1. Is the perception of software as “soft” and therefore easy to change responsible for so many requirements changes that it interferes with the chances of software projects succeeding? (See Point 1.) Has anyone developed good metrics to “push back” against such changes, comparable to “long lead” hardware items and expensive materials?

2. Software requirements and test are often managed in a rigorous way, such that nothing is allowed to continue as a software requirement that is not testable in a reasonable way. (See Points 1 and 2.) Test cases are restricted to those that directly trace to one or more of these filtered requirements. How should goals of users, commonly unclear and untestable but real, be handled? Do such filtering practices contribute to software systems being considered unusable by users even though they can be demonstrated to meet requirements?

3. Is software development too silo-oriented? Is there too much reliance on speaking through documents, such as requirements documents (systems engineers to coders) and interface documents (one module coder to the next)? (See Points 4 and 5) If so, why, and what can be done to improve the integration of software engineers?

4. Does the emphasis on process, particularly that encouraged by discipline-specific instruments such as the software CMM, inhibit integrated product and process development (IPPD)? Would this be improved, or made worse, if disciplines created their own CMMs, as some are doing? (See Point 9)

5. Do software systems (See Point 2) develop complexity faster because they are not “well-designed”? (Or, perhaps, because software is perceived to be able to do anything, or because the requirements change rapidly?) Is rapid development of complexity something that can be corrected through good design practices? Can systems engineering practices aid in the resolution of complexity problems in software?

6. Does software need to know the final constraints that hardware will impose upon it before software design is begun? Why would this be any different from hardware, which would want the final constraints software would impose upon it before hardware design would be started? A study comparing the schedule effects of technology insertion into various kinds of system development programs would be very useful.

7. Is software “just another component” or something more than a hardware component? Are some of these arguments repeats of arguments that were held many years ago when the last new technology took hold of systems? (Perhaps transistors?) If so, what happened then, was it the right thing, and should it be encouraged here?

8. How do we better integrate software and hardware disciplines, with each other, and with systems engineering?

CONCLUSION

Interesting counterpoints brought up by respondents, of backgrounds ranging from computer science to hardware, bring to bear the advantages of working in a team of people with diverse skills -- no one person could have provided so many interesting views and examples.

In general, software and hardware systems do commonly need somewhat different systems engineering. One difference that will not change is related to the logical nature of software and physical nature of hardware. Other variations are related to how preceded the system is, or how complex or how large, or to what extent COTS objects are being incorporated. Software system complexity comes in part from the ability of software to solve user problems never addressed by hardware systems, and their resultant implicit acceptance of the complexity of the problem space.

With time, many of these differences will decrease. Software will evolve as a discipline, and process documentation will likely become more useful and less intrusive. Software COTS objects will become increasingly available, and with more software systems, the chance of having precedents increases. Finally, research may answer some of the open questions above and further improve the engineering of systems of all types.

ACKNOWLEDGEMENTS

Jim Armstrong, Cecilia Haskins, Howard Lykins, and Bill Schoening provided encouragement and suggestions, and Bill provided the idea for the “Open Questions” section as well as a head start on several of the questions. Without these people, and the twenty list

---

8 (Sheard 1998) discusses the field of systems engineering standards. The international systems life cycle standard, ISO 15288, is being developed out of the software standards committee, because there is no comparable systems engineering standards committee. One major quandary is whether the systems standard should work around the software standard, ISO 12207, primarily discussing “everything else,” or whether the systems standard should take a top-down approach, and later revise the software standard to clearly define the interfaces between the two. Similar issues plague the maturity model and software standards fields. In any event, the answers will not be simple, or simple to implement.
server respondents, this paper would have remained just the inkling it has been since its main idea was first conceived at the INCOSE 1994 symposium. And finally, Claire Brown and Lisa Finneran had the courage to tell a bullheaded author when the paper “wasn’t good enough.”

REFERENCES
Courtney, Steve. incose-discuss list server post, 1997-09-09.
Cutler, William H. List server post, 1997-09-09.
Gabb, Andrew. List server post, 1997-09-10.
Gurwell, Noll. List server post, 1997-09-09.
Hongsermeier, Martin. List server post, 1997-09-10.
Hyman, Kevin. List server post, 1997-09-10.
Kasser, Joseph. List server post, 1997-09-09.
Lister, Peter. List server post, 1997-09-10.
Owens, W. R. List server post, 1997-09-09.
Schroeder, Curt. List server post, 1997-09-11.
Tockey, Stephen R. List server post, 1997-09-11.
van Gaasbeek, James. List server post, 1997-09-09.
Wittig, Bill. List server post, 1997-09-09.
Zittel, Randy. List server post, 1997-09-10.

BIOGRAPHY
Sarah A. Sheard has eighteen years’ experience in systems engineering, including hardware systems (satellites) and software systems such as air traffic control. Currently, Ms. Sheard develops and teaches systems engineering-related products including guidebooks and courses, helps companies improve their systems engineering efforts and helps integrate software and systems engineering efforts. Ms. Sheard received a BA in Chemistry from the University of Rochester in 1976 and an MS in Chemistry from the California Institute of Technology in 1979.