A Rationale and Framework for Integrating Systems Engineering and Software Engineering

Richard Turner, Arthur Pyster, Michael Pennotti
Stevens Institute of Technology
Castle Point on Hudson
Hoboken, New Jersey, USA 07030
{richard.turner, art.pyster, michael.pennotti}@stevens.edu

Copyright © 2009 by Richard Turner, Arthur Pyster, Michael Pennotti. Published and used by INCOSE with permission.

Abstract. As illustrated in the harmonization of ISO 15288 and ISO 12207, the disciplines of systems engineering and software engineering share many processes, methods, and concerns. In most systems of interest, software now provides a significant majority of functionality and increasingly is a significant component in determining non-functional attributes. Failures in system acquisition and implementation are often attributed to systems and/or software engineering shortfalls. The need to apply systems engineering to software design and development, as well as the need to incorporate software engineering concepts into systems engineering decisions, implies that a closer relationship between the two disciplines is appropriate. This paper states a clear rationale for integrating the two disciplines, defines a framework that establishes a vocabulary for discussing integration, and presents initial findings in using that framework to describe systems and software integration in ongoing defence programs.

From the Trenches

> "The selected architectural concept resulted in underutilized software capability."
>
> "Non-functional requirements, such as reliability, cause difficulty when there is software involved. Hardware reliability numbers are calculated to many decimal places, and include the contributions of very low-level WBS [work breakdown structure] components, but software reliability is not understood and so ignored."
>
> "Unless developed collaboratively, software specifications often limit the software architects and designers trade space for solutions."
>
> "We [software developers] often find ourselves having to rewrite the software specifications provided by systems engineering so that they are achievable."
>
> "The difference in speed of maturation between hardware [system] requirements, and software requirements causes tension between systems engineers and software engineers."

These are a few nuggets from interviews on integrating systems and software engineering conducted with nine defence acquisition programs. Most of the programs interviewed were successfully dealing with integration issues in individual, context-related ways. They all agreed, however, that the systems being developed today needed more collaboration between software and systems engineers.

Of course, this raises issues as to the definition of collaboration and how much of it is needed. Unfortunately, because of poor communication concepts, there has not been a good way to discuss these and related issues.

The Evolution of Systems

The meaning of "system" has changed over time as technology and the technological infrastructure have changed. Obviously, there have always been systems, but the way we as humans perceive them has evolved. To early humans nature was a system totally dependent on the wishes and whims of local deities or spirits. This is a long way (or perhaps not) from our partial understanding of the intricate and complex interactions of the earth's ecosystem. The systems we are primarily concerned with are those that are human-engineered, and as you would imagine, their definitions have changed as well.

Systems engineering (SE) was developed to address systems we refer to by the retronym ***classical systems***. Originally composed purely of hardware, these systems evolved to include software embedded within the hardware components. As hard as it might be to imagine today, the gun directors and bombsites of World War II were analog computers that mechanically modeled the computational problems they were intended to solve; passenger cars were devoid of software until the late 1960s; and electromechanical crossbar switches formed the backbone of public switched telephone networks well into the 1970s. When software was introduced into these systems, it was generally treated as another component, a simple extension of the hardware. This treatment was appropriate, since the limitations of the available technology ensured that software was relegated to low-level functionality, deeply embedded within the hardware and used to perform functions that were incidental to the interests of systems engineers.

Software engineering (SwE) was developed to address the design of ***information systems*** in which software was the principal source of functionality and existed at the highest levels of the system. Through the use of layered architectures, these systems were designed in such a way as to insulate the application software as much as possible from the general-purpose hardware on which they operated. In effect, software in these systems used the hardware but it was designed so it could be ported from one hardware platform to another and modified without regard to the hardware substrate beneath it.

Today's systems, however, from cell phones and automobiles to aircraft and ammunition, are neither primarily hardware nor primarily software, but rather a highly interwoven combination of both. Consider the case of automotive anti-skid, in which an automobile's steering, brakes, throttle and active suspension "collaborate" to ensure that the vehicle maintains traction under the most hazardous of circumstances. The success of this function does not depend primarily on hardware or primarily on software, but on the close cooperation of both working together without the driver in the loop. At a recent meeting, a speaker asserted that "the only system that doesn't contain software today is bullets," to which another participant responded, "Oh, we make bullets with software."

The truth is that today, nearly all interesting systems are a combination of hardware and software, in which both are equal partners in realizing the required system functionality. For these ***interdependent systems***, neither traditional SE techniques nor traditional SwE techniques are sufficient by themselves – the hardware and software must be designed in an integrated fashion (Doyle and Pennotti 2006). Questions such as how to flow non-functional requirements down to components, how to predict system behavior, and how to verify system performance are complicated by the interrelationship and interaction between hardware and software.

The three types of systems are illustrated in Figure 1.

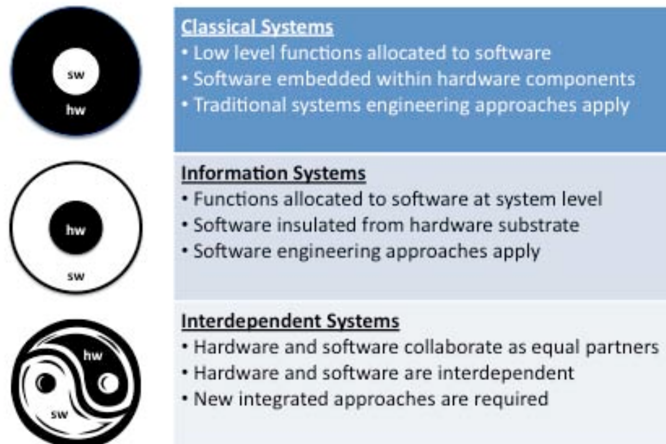

Figure 1. The three types of systems

The traditional response to suggestions that SE should be integrated with SwE is that SwE is just another specialty engineering discipline. At a briefing of some of these concepts to the Corporate Advisory Board at the 2008 INCOSE conference in Utrecht, several systems engineers strongly stated that this was indeed the case, and that the integration of SE and SwE should be the job of the chief systems engineer. Others suggested that it was more appropriate to include software as only one of several disciplines that should be "integrated." Not surprisingly, the one point upon which there was general agreement, was that SwE certainly needed more SE discipline.

To all these comments, except for SwE needing more SE, we respectfully disagree. While the comments may have been true five or more years ago, they reflect relationships fundamentally different from those in the interdependent systems currently being created and envisioned. We assert the following:

- Interdependent systems are those where:
  - A "major" portion of the capabilities/value of the system is delivered through software
  - A "major" portion of system quality attributes "largely" depend on software (safety, security, agility, reliability, availability, resilience, ...)
- Today almost all high value systems meet these criteria and the percentage is increasing
- In such interdependent systems, almost all important decisions require equal consideration of SwE and SE expertise, including those associated with technical, management, personnel and customer concerns.
- Decision making is supported by information drawn from the experience and competencies of the decision makers and the execution of engineering and management processes. Critical decisions in interdependent systems are made most effectively by people who have strong SwE and SE competencies and experience and who have information from both SE and SwE processes.
- Left as independent processes, systems and software decisions are more likely to be made without complete understanding or in temporal displacement leading to poor results or missed opportunities downstream.

In a world of interdependent systems, practicing SE and SwE as completely distinct disciplines is *harmful*. The requisite security, safety, maintainability, and other critical attributes of high-value, high-quality systems demand that the interplay between these two

disciplines is well understood, measured, and continuously improved. Therefore, we believe that it is essential that the SE and SwE disciplines, as represented by their bodies of knowledge and processes, be rapidly and effectively integrated.

Software Engineering as Pioneer

Before considering the challenges to integration, it may be useful to consider how SwE evolved as a discipline. In his keynote address to the initial convocation of the University of Southern California's Center for Systems and Software Engineering, William Wulf, past president of the National Academy of Engineering, made the observation that systems engineers should look at SwE for guidance in adapting to the new realities of system design. He pointed out that with custom-engineered materials, nanotechnology and other advances, the "near-infinite, non-physically limited" solution space of software may be on the cusp of applying to systems.

In his ICSE 2006 keynote paper "A View of 20th and 21st Century Software Engineering," Barry Boehm provided a detailed description of SwE history (Boehm 2006). He describes the evolution of the discipline in term of Hegel's cycles of human understanding: ideas (thesis), different ideas when the original doesn't quite work (antithesis), and hybrid solutions (synthesis). Table 1 summarizes this brilliant paper.

Table 1. Boehm's Hegellian History of Software Engineering

Decade	Concept	Description
1950's	Thesis: Software Engineering Is Like Hardware Engineering	High cost of computer time (300 times developer's salary); software developers were mathematicians, physicists and hardware engineers; software functionality primarily algorithms
1960's	Antithesis: Software Crafting	Ease of modification led to code and fix; more people-intensive systems; software reliability and maintenance differ from hardware; developers no longer scientists and engineers – "cowboy programmers" become role models
1970's	Synthesis and Antithesis: Formality and Waterfall Processes	Reaction to synthesize hardware techniques into software engineering; Royce's waterfall diagram; quantitative approaches; short-lived formal approaches; software costs exceeding hardware costs
1980's	Synthesis: Productivity and Scalability	Initiatives to address 70's problems; Standards (DoD-STD-2167, MIL-STD-1521, ISO-9000); Capability Maturity Model (SW-CMM); integrated tool environments; software factories; reuse approaches; "No silver bullet" paper
1990's	Antithesis: Concurrent vs. Sequential Processes	Emphasis on time-to-market; COTS; concurrency of engineering activities; I'll Know It When I See It (IKIWISI) requirements; risk-driven spiral; Rational Unified Process; Open Source; emphasis on human-computer interface
2000's	Antithesis and Partial Synthesis: Agility and Value	Pace of change accelerates; frustration with heavy process leads to agile approaches and value-based SwE; software ubiquity and criticality raises dependability issues; COTS, Open Source and legacy software become significant productivity drivers; model-based development paradigm; realization of SE/SwE interactions

As you can see, SwE has had a colourful and often amnesiac evolution. Much less than SE, SwE has had to significantly change its mental models and goals to cope with rapidly changing technology capabilities, infrastructure and system expectations. One good reason for integrating SE and SwE is to take advantage of these experiences and avoid some of the historical software engineering mistakes.

Challenges to Integration

Historical and cultural barriers and technical issues make integrating the two disciplines challenging. Historical context and vestigial prejudices are significant barriers. SE and SwE cultures are significantly different due to the personality, background and approaches of practitioners. The normal educational background for each discipline is significantly different. Systems engineers are generally developed out of traditional engineering disciplines and so have more of the traditional engineering knowledge and skills (thermodynamics, control theory, stresses, …). Software engineers, although often drawn from mathematics or computer science curricula, are more likely to have been initially trained in a liberal arts and sciences program. The software viewpoint is based on an unlimited, essentially totally malleable solution space while systems engineers are limited by the laws of physical science. The correctness of complex software is essentially un-provable while systems engineers constantly wrestle with assuring safety, reliability and availability in deterministic environments.

These barriers lead to four fundamental issues that must be addressed in order to successfully integrate these disciplines.

Issue 1: *Vocabulary.* There is no precise way to talk about the integration of systems and software engineering. We need to make old terms precise and introduce new terms to enable clear conversations. Three examples illustrate the need for clarity:

1. The phrase *integration of systems and software engineering* itself has no precise meaning.
2. Object-oriented methods for architecting software clash with structured methods common for architecting systems. This is a clash in mental models. Systems and software engineers think about architecting differently.
3. When companies independently define competency models for their SE and SwE workforce, that independence aggravates a communications gap between them.

Introducing terms such as *clash* and *gap* in a well-defined way is essential to understanding the integration of SE and SwE.

Issue 2: *Measurement.* There is no precise way to talk about *how much* integration there is between SE and SwE in a particular situation. Without a way to quantify the amount of integration, it is hard to understand what to change and what the impact of a change is. For example, if a company introduces a requirement that all systems engineers have at least a journeyman's knowledge of SwE, have they added a "little integration" or "a lot"?

Issue 3: *Entanglement*. There is no way to simultaneously understand the many ways in which the SwE and SE disciplines touch. SE and SwE are large complex disciplines. For example, the IEEE Software Engineering Body of Knowledge, which defines the structure and top-level content of SwE, is more than 200 pages long. (Abran, et al 2004). The INCOSE SE Body of Knowledge is similarly long and complex (INCOSE 2004). A way to simultaneously decompose SE and SwE is required; e.g., by using ISO 15288 (ISO/IEC 2008) which defines 25 different high-level processes that both disciplines use.

Issue 4: *Value.* There is no comprehensive list of benefits that can be achieved by integrating SwE and SE nor is there an understanding of the associated costs.

Achieving and maintaining integration will not be either easy or free. The associated set of costs must be understood so the business case for integration can be made and integration activities can be prioritized by the value they deliver.

Touchpoint: A Framework for Describing Integration

A framework is a conceptual structure used to address complex issues. Our framework, *Touchpoint*, focuses on the issues surrounding the integration of SwE and SE for interdependent systems. As shown in Figure 2, our framework has four primary components: *Processes*, *Touchpoints*, *Faults*, and *Resolution Strategies*.

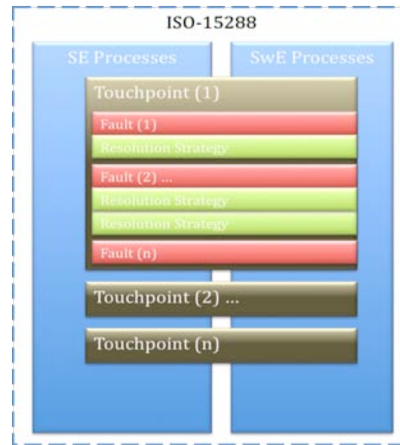

Figure 2. The Touchpoint Framework

Processes. These are the ordered activities that define the systems and software engineering disciplines. As identified in Issue 3, SE and SwE are each large complex disciplines. Fortunately, the new version of ISO 15288 provides a common way to structure SE and SwE via 25 processes, gathered into four process groups as shown in Table 2.

Table 2. The 25 ISO 15288 Processes

Process Group	Process	Process Purpose
Agreement	Acquisition	Obtain a product or service in accordance with the acquirer's requirements
	Supply	Provide an acquirer with a product or service that meets agreed requirements
Organizational Project-Enabling	Life Cycle Model Management	Define, maintain, and assure availability of policies, life cycle processes, life cycle models, and procedures for use by the organization with respect to the scope of 15288
	Infrastructure Management	Provide the enabling infrastructure and services to projects to support organization and project objectives throughout the life cycle
	Project Portfolio Management	Initiate and sustain necessary, sufficient, and suitable projects in order to meet the strategic objectives of the organization
	Human Resource Management	Ensure the organization is provided with necessary human resources and to maintain their competencies, consistent with business needs
	Quality Management	Assure that products, services, and implementations of life cycle processes meet organization quality objectives and achieve customer satisfaction
Project	Project Planning	Produce and communicate effective and workable project plans
	Project Assessment and Control	Determine the status of the project and direct project plan execution to ensure that the project performs according to plans and schedules, within projected budgets, to satisfy technical

Table 2. The 25 ISO 15288 Processes

Process Group	Process	Process Purpose
		objectives
	Decision Management	Select the most beneficial course of project action where alternatives exist
	Risk Management	Identify, analyze, treat, and monitor the risks continuously
	Configuration Management	Establish and maintain the integrity of all identified outputs of a project or process and make them available to concerned parties
	Information Management	Provide relevant, timely, complete, valid, and, if required, confidential information to designated parties during and, as appropriate, after the system life cycle
	Measurement	Collect, analyze, and report data relating to the products developed and processes implemented within the organization, to support effective management of the processes, and to objectively demonstrate the quality of the products
Technical	Stakeholder Requirements Definition	Define the requirements for a system that can provide the services needed by users and other stakeholders in a defined environment
	Requirements Analysis	Transform the stakeholder, requirement-driven view of desired services into a technical view of a required product that could deliver those services
	Architectural Design	Synthesize a solution that satisfies system requirements
	Implementation	Realize a specified system element
	Integration	Assemble a system that is consistent with the architectural design
	Verification	Confirm that the specified design requirements are fulfilled by the system
	Transition	Establish a capability to provide services specified by stakeholder requirements in the operational environment
	Validation	Provide objective evidence that the services provided by a system when in use comply with stakeholders' requirements, achieving its intended use in its intended operational environment
	Operation	Use the system in order to deliver its services
	Maintenance	Sustain the capability of the system to provide a service
	Disposal	End the existence of a system entity

An organization developing, maintaining, or disposing of an interdependent system executes these 25 SE processes and 25 SwE processes. Of course, the organization may not document, explicitly acknowledge, consistently or adequately perform these processes – but they do perform them. Each process contains constituent activities that are executed by engineers, analysts, managers, and others.

Touchpoints. We say that two processes *touch* when interactions between their constituent activities affect program risk or value – positively or negatively. We call this a *touchpoint*. For example, properly verifying an interdependent system requires verifying the system's software. If a system verification process has explicit activities that interact with software verification activities, then the systems verification process and the software verification process touch and we can define at least one touchpoint for each "touching" activity.

Faults. Touchpoints are not always implemented well. A touchpoint may exist, but the process or activity may fail to produce its maximum value. The framework defines these shortcomings as *faults*. Three types of faults are defined in Table 3.

Table 3. Three Types of Faults in SE/SwE Integration

Fault Type	Description
Gap	Logically, there should be an interaction between the corresponding SE and SwE processes, but the processes do not include one. A needed activity is therefore performed poorly, or not performed at all.
Clash	One or more activities in each of the two corresponding SE and SwE processes produce are incompatible and result in inconsistent results or inconsistent actions.
Waste	Activities in the two corresponding SE and SwE processes independently expend resources that produce the same result or take the same action with no added benefit to the program

A poor system verification process could (1) pretend that software errors do not affect system correctness, or (2) systematically underestimate the time it will take to properly test software or the complexity of such testing, or (3) independently redo the same testing that the software verification process does. Each of these examples is a *fault* in the integration of SE and SwE, which we associate with the system and software verification processes. The first fault would be a *gap* between the two processes. The processes should touch, but do not because those who define or execute the processes do not understand the need. Consequently, important verification activities will not be performed. The second fault would be a *clash* between the two processes. Each process has its own estimation method, and those methods produce inconsistent results. The third fault would be a *waste* between the two processes. Clashes can be further characterized as resulting from differences in *vocabulary, value,* or *mental model,* as shown in Table 4.

Table 4. Types of Clashes

Clash Type	Description
Vocabulary	Touching SwE and SE processes or their constituent activities use the same terminology with different meanings, or terms not recognized by the other, making communication between software and systems engineers harder; e.g., traditional SE requirements methods are not object-oriented and therefore do not include such terms as "inheritance" or "class hierarchy," both of which are common in SwE object-oriented requirements methods.
Value	Software and systems engineers in an organization or program value different process characteristics; e.g., a systems engineer may value a stable baseline while a software engineer may value a rapidly evolving and flexible baseline.
Mental Model	Software and systems engineers think differently about how to carry out process activities; e.g., systems architecting methods most often define "part-of" relationships while software architectures primarily define "uses" relationships.

Touchpoints and their faults are specific to an organization or project's performed process. However, as shown in Table 5, it is possible (and valuable) to identify and describe some relatively common process touchpoints and their faults.

Table 5. Example Touchpoint

Process	Touchpoint	Fault	Type
Architectural Design	Systems architectures include significant software components to deliver critical capability	Software architectures define layers of related functionality, while most methods for systems architectures create hierarchical structures.	Clash – Mental Model

Resolution Strategies. Naturally, when faults are unearthed, there is a desire to fix them, especially those with high impact on risk or value. For each fault, there may be one or more resolution strategies, which, when executed well, will eliminate the fault or at least reduce its impact. In some cases, resolution strategies are known and just need to be applied; on the other hand, resolving some faults will require research.

In the Touchpoint framework, resolution strategies are grouped into four traditional categories: *process, people, environment,* and *technology*. Any number of resolution strategies in each category is possible for a fault. Table 6 presents resolution strategies for the touchpoint/fault in Table 5. The first resolution strategy is the opinion of the authors based on their understanding of the field. The last two resolution strategies, shown in italics, came directly from interviews with practitioners on programs that had used those strategies successfully.

Table 6. Resolution Strategies

Process	Touchpoint	Fault	Type
Architectural Design	Systems architectures include significant software components to deliver critical capability	*Software-engineering architectures define layers of related functionality, while most systems-engineering methods are hierarchical structures.*	Clash – Mental Model
Resolution Strategy			**Category**
Research must be conducted to resolve the clash between object-oriented and structured methods. Maier provides some of the best research in this area.			Technology
Design software architecture to look just like system architecture. Make it easy for a system architect to understand. (SW systems mirror HW systems, e.g. relays, motors, etc). Then SW helps the system architect understand things in better detail.			*Process*
Middleware may be able to bridge the gap.			*Technology*

Applying the Framework: Early Pilot Experience

The authors piloted Touchpoint to explore its explanatory value. A stated earlier, while touchpoints may be defined generically, they exist in the actual SE and SwE processes on real programs. Since most activities with possible touchpoints occur at a level below much of

the typical process documentation, pilot activity needed to engage practitioners to identify touchpoints. To this end, 1-hour interviews were conducted with a group of systems and software engineering leaders for each of 9 defence programs across a variety of domains, complexity, size, and developer. We also interviewed a key member of the Office of the Secretary of Defence (OSD), which has performed systemic analysis of data collected during more than 50 program reviews.

In the analysis of the interviews, touchpoints were identified in three ways. First, something was described as a touchpoint by one of the interviewees. Second, a problem might have surfaced that implied the existence of a touchpoint. Third, an interviewee may have described a strategy taken when solving a specific problem that implied a touchpoint. Table 7 is an overview of the touchpoints identified, organized by issue category. In counting the number of projects, the interview with OSD was considered a single project. The complete set of touchpoints can be found in Appendix A [**<To be supplied in final paper>**].

Table 7. Summary of Touchpoints

Category	Touchpoints	No. of Projects
Architecture	12	6
CM	1	1
EVM	2	2
Human Capital	4	2
Process Planning	3	3
Requirements	23	10
Risk Management	2	2
System Integration	4	4
Software Metrics (Visibility)	4	3
Contracting	4	3
Life Cycle	7	4
Technical Reviews	2	2

Conclusions and Recommendations. In general, the framework did a good job of capturing real experiences from programs and provided a reasonable vocabulary to discuss integration of the two disciplines. However, we believe that the data collected so far only scratches the surface of what is required. While we did identify a good number of touchpoints that could be assigned to the various categories, the limited amount of time available for interviews often resulted in a conversation focused on only one or two areas. Moreover, even with clear non-attribution, it was frequently difficult for interviewees to discuss ongoing problems in an interview, leading to a preponderance of identifications based on resolution strategies, rather than on issues.

References

Abran, Alain, et al. “Guide to the Software Engineering Body of Knowledge (SWEBOK),” IEEE Computer Society, 2004

Barry Boehm, "A View of 20th and 21st Century Software Engineering," *Proceedings of the 28th International Conference on Software Engineering*, International Conference on Software Engineering, 2006, pp. 12-29.

Doyle, L. and M. C. Pennotti. “Impact of Embedded Software Technology on Systems Engineering,” 16th Annual International Symposium, INCOSE, 2006.

INCOSE, *Guide to Systems Engineering Body of Knowledge*, International Council on Systems Engineering, 2004.

ISO/IEC, *Systems and software engineering – System lifecycle processes*, ISO/IEC 15288:2008, International Organization for Standardization, 2008.