AC 2011-2503: TEACHING SOFTWARE ENGINEERING TO UNDERGRADUATE SYSTEM ENGINEERING SUDENTS

Richard Fairley & Mary Jane Willshire, Software and Systems Engineering Associates

Richard E. (Dick) Fairley is founder and principal associate of Software and System Engineering Associates (S2EA; a consulting and training company) and an adjunct professor at Colorado Technical University in Colorado Springs, Colorado. Dr. Fairley has bachelors and masters degree in electrical engineering. His PhD in computer science is from UCLA. He can be contacted as d.fairley@computer.org.

Mary Jane Willshire is a principal associate of S2EA. Dr. Willshire has bachelors and masters degrees in mathematics. Her PhD in computer science is from Georgia Tech. She can be contacted as mjwillshire@ieee.org.

Teaching Software Engineering Concepts to Systems Engineering Students

Abstract

This paper describes the software engineering concepts that systems engineering students need to understand in order to effectively work with software engineers who may be members of their system engineering teams, both as students and as practitioners. Ways to introduce this material into systems engineering curricula are addressed. This paper is a companion to "Teaching Systems Engineering to Software Engineers" which appears in the conference proceedings of the 2011 Conference on Software Engineering Education and Training, published by ACM/IEEE.

Introduction

It is widely recognized that software is the element that binds together the diverse components in most modern systems and provides much of the functionality in those systems. The following quote from draft version 0.25 of the emerging body of knowledge for systems engineering makes this point[1]:

> "Virtually every interesting system today has significant software content. In fact, most of the functionality of commercial and government systems is now implemented in software and software plays a prominent, often dominant, role in differentiating competing systems in the marketplace.
>
> Software engineering (SwE) is not just an allied discipline to systems engineering (SE). SwE and SE are intimately entangled. Software is usually prominent in modern systems architectures and is often the glue for integrating complex system components."

Many systems engineering students are never exposed to software engineering other than, perhaps, through an introductory programming class. The role of a systems engineer is to orchestrate and coordinate the diverse disciplines that may be required to develop a complex system. Thus, systems engineers do not need to know how to write computer programs (i.e., the details of software construction) any more than they need to know how to fabricate a special purpose computer chip or design a power supply. They do need to understand the processes, procedures, parameters, and constraints under which software engineers design and build software in order to effectively communicate with them and work with them. To a systems engineer, software engineers are one of several, and possibly many different kinds of system specialists with whom they must work.

This paper presents some of the most important topics that systems engineers need to understand about software engineering including:

- Differences in use of shared terminology
- System engineering techniques used by software engineers
- Software engineering techniques used by systems engineers
- The intangible and malleable nature of software
- The four essential properties of software
- The three additional factors
- Risk management of software projects
- Software development processes

For purposes of exposition, we distinguish software engineering from software construction. Software engineers are concerned with analysis and design, allocation of requirements, component integration, verification and validation, re-engineering of existing systems, and life cycle sustainment of software. Programmers, who may also be capable software engineers, construct software (i.e. engage in detailed design, implementation, and unit testing of software). Software engineers work with software component specialists (e.g., user interface, database, computation, communication specialists) to construct or otherwise obtain the needed software components. In some cases, software engineers may be component specialists for certain kinds of components; they engage with other kinds of software component specialists as necessary.

Software engineers are often involved in managing the technical aspects of a project or program in the same way that systems engineers may manage the technical aspect of a systems project or systems program. Thus, software engineers need to understand how project management techniques, such as those included in the Project Management Institute's Body of Knowledge (PMBOK®) must be adapted for software projects[2,3].

These commonalities would make it appear that software engineering is merely an application of systems engineering; however, this is only a surface appearance. Systems engineers need to understand how these similar-sounding work activities are different in the software domain from those in other engineering disciplines.

The differences arise from the intangible nature of software and the physical nature of other engineering artifacts. This results in different approaches to curriculum design and different approaches to problem solving, which in practice results in different cultural attitudes, different uses of terminology, and different communication styles.

Table 1, below, and some of the accompanying text also appears in the paper "Teaching Systems Engineering to Software Engineers," published in the proceedings of the 2011 Conference on Software Engineering Education and Training (CSEET 2011).

It is somewhat ironic that there should be a disconnect between system engineering education and software engineering education and the practice of the two disciplines because many of the concepts of software engineering have been adapted from system engineering, including stakeholder analysis, requirements engineering, functional decomposition, design constraints, architectural design, design criteria and design tradeoffs, interface specification, traceability, configuration management, and systematic verification and validation. However, the lack of physical properties and the resulting malleability of software has resulted in differences in the ways the methods and techniques are applied at the system level and at the software level.

It is also the case that some methods developed by software engineers, such as model-driven development, UML-SYSML, use cases, object-oriented design, agile methods, continuous integration, incremental V&V, and process modeling and improvement can be, and are being used by system engineers. However, systems engineering and software engineering students need to understand the similarities and differences in the ways in which these concepts are applied in each discipline.

In addition, the nature of software and the methods used to develop software have resulted in different uses of terminology and emphasis on different aspects of the artifacts produced by system engineers and by software engineers.

Some examples: the term “performance” is often used by software engineers in the narrow sense of throughput and response time, whereas systems engineers often take performance to mean satisfaction of all the non-functional requirements for a system. In software engineering, the term “baseline” is used to denote any work product that has been determined to be acceptable and is placed under change control. Software baselines are more fluid, and change more frequently than baselines of physical entities because the software representation is in the same medium as the software that controls the baseline and because software being developed or modified is frequently updated; often on a daily basis.

Use of off-the-shelf manufactured components is routine in systems engineering of physical systems. In software engineering, use of existing components is termed “reuse.” Components to be reused may be obtained from an open source on the Internet, from a software vendor, from a corporate library, or from a programmer’s private library. The flexible nature of software may facilitate some comparatively easy modifications to the component to be reused, as compared to modifying a physical component; however, software reuse is not “free.” Candidate components must be identified, evaluated, perhaps modified, and integrated into the software system.

Safety and dependability are sometimes addressed in software engineering curricula; however, students of both systems engineering and software engineering need to understand how these issues can and cannot be addressed at the software level. In addition, systems engineering students need to understand how software engineers view non-functional considerations such as cost/benefit tradeoffs, risk management, reliability, and MTTF. These issues are summarized in Table 1.

Table 1.
Systems Engineering Methods adapted to Software Engineering (SE to SwE) and Software Engineering Methods adapted to Systems Engineering (SwE to SE)

1a. SE to SwE

stakeholder analysis
requirements engineering
functional decomposition
design constraints
architectural design
design criteria
design tradeoffs
interface specification
traceability
configuration management
systematic verification and validation

1b. SwE to SE

model-driven development
UML-SYSML
use cases
object-oriented design
iterative development
agile methods
continuous integration
incremental V&V
process modeling
process improvement

Disconnects between systems engineering and software engineering occur for two primary reasons: 1) the contrasting nature of the intangible software medium and the physical media of traditional engineering, and 2) because systems engineers and software engineers receive pedagogically different educations.

Concerning primary reason 1: Software is an intangible artifact. Design documents and source code are representations of software but they are not the software; the actual software resides in the current flows and magnetizations of an enormous number of computing-device elements. Software does not always behave in the expected ways because the software developers may have misunderstood how the source code, as written, is translated into executable object code and how the hardware will decode and execute the object code (i.e., the translated source code). In contrast, the systems produced by traditional engineers consist of physical structures and machines that have tangible properties. Software has no tangible properties: it cannot be directly seen, smelled, tasted, felt, or heard. Only the resulting behavior of physical devices, as controlled by the internal object code, can be observed.

Concerning primary reason 2: Most systems engineers are educated firstly as engineers and secondly as systems engineers. This means that most systems engineering curricula include a traditional undergraduate core of engineering fundamentals, which is based on continuous mathematics and engineering problem solving. Many practicing systems engineers were educated in a traditional engineering discipline, rather than systems engineering, and became systems engineers through job experiences in developing systems where the primary focus was on physical components. Both approaches (education and experience) result in systems engineers who are firmly grounded in traditional engineering.

In contrast, software engineering curricula are based on computer science, which emphasizes discrete mathematics and algorithmic problem solving. Many practicing software engineers (like many practicing systems engineers) may have received little education in software engineering (or systems engineering) and have obtained their skills through practical experience.

The remainder of this paper expands on the similarities and differences between software engineering and traditional engineering. Software engineering issues that should be understood by systems engineers are then presented and different approaches to incorporating software engineering concepts into systems engineering curricula are discussed.

What's Different About Software?

Software, in contrast to other products of engineering, does not have any physical properties, other than the printouts and screen images of design documents and source code. In Chapter 16 of his seminal text, *The Mythical Man-Month*, Fred Brooks identified four essential properties of software that differentiate it from other kinds of engineering artifacts[4]:

1) complexity,
2) conformity,
3) changeability, and
4) invisibility of software.

An exposition of these properties is provided in Chapter 1 of *Managing and Leading Software Projects*[3]. The following is a brief paraphrasing from that text.

Software complexity

According to Fred Brooks, software is more complex, for the amount of effort and the resources required to construct it, than most artifacts produced by similar amounts of effort and resources. Clearly, products and systems that contain software are more complex than the software within them but those products and systems, including the software, require additional effort and resources to develop the other elements.

The complexity of software arises from the large number of unique, interacting components in a software system. The components are unique because, for the most part, they are encapsulated as functions, subroutines, or objects and invoked as needed rather than being replicated. Software components have several different kinds of interactions, including serial and concurrent invocations, state transitions, data couplings, and interfaces to databases and external systems. In addition, the algorithms and data structures contained within the components may be complex.

Complexity internal to the components and in the connections among components, plus the ripple effect of changes to a component may result in a large amount of rework that can result from a "small" change in requirements. For this reason, many experienced software personnel say there are no small changes to software requirements. Complexity can also hide defects that may not be discovered immediately and may thus require additional, unplanned rework later. The move to object-oriented development of software is motivated, in part, by the desire to encapsulate algorithms and data structures behind well-defined interfaces, thus mirroring the "black box" approach to systems design used by traditional engineers.

Software conformity

Conformity is the second essential property of software cited by Brooks. Software must conform to exacting specifications in the representation of each part, in the interfaces to other internal parts, and in the connections to the environment in which the software operates. A compiler that translates source code to object code can detect a missing semicolon or other syntactic errors but a defect in the program logic, or a timing error during program execution may not be detected during software development or modification, and the source of the resulting undesired behavior may be difficult to detect when encountered during system operation.

Software conformity in the interfaces between software components is also an issue; subtle defects in interfaces are one of the leading causes of software failure. Tolerances among the interfaces of physical entities is the foundation of manufacturing and construction; no two physical parts that are joined together have, or are required to have, exact matches. Eli Whitney (of cotton gin fame) realized in 1798 that if musket parts were manufactured to specified tolerances, interchangeability of similar (but not identical) parts could be achieved.
There are no corresponding tolerances in the interfaces among software entities or between software entities and their environments. Interfaces among software parts must agree exactly in the numbers and types of parameters and in the kinds of couplings. There are no interface specifications for software stating that a parameter can be "an integer plus or minus 2%."

Also, lack of conformity can cause problems when an existing software component cannot be reused in a different system because it does not conform to the needs of the system under development. Lack of conformity might not be discovered until late in a project, thus necessitating development and integration of an acceptable component to replace the one that cannot be reused. In addition, complexity of the candidate component or complexity in the design of the system being developed may have made it difficult to determine that the component to be reused lacked the necessary conformity until the components it would interact with were implemented.

Software changeability

Changeability is Brooks' third property that makes software development difficult. Software coordinates the operation of physical components and provides much of the functionality in software-intensive systems. Software is the most frequently changed element in a system that contains software because it is easily changed (i.e., the most malleable) as compared to modifying physical components. This is especially the case in the latter stages of developing or enhancing a system. Changes may occur because customers change their minds; competing products change; mission objectives change; laws, regulations, and business practices change; underlying hardware and software technology changes; the operating environment of the system changes. As mentioned above, software may be the most readily changed component but that does not mean changes to software are easily done.

When a system is installed in the operating environment it will change that environment and result in new requirements that will require changes to the system; i.e., now that the new system enables me to do A and B, I would like for it to also allow me to do C, or to do B in a different way, or to do C instead of B. Often, changing the software is the most cost-effective way to make changes to a software-intensive system; but as stated above there are no small changes to complex software.

Software invisibility

The fourth of Brooks' essential properties of software is invisibility. Software is said to be invisible because it has no physical properties. Because software has no physical presence, other than the design and source code representations, software engineers use these representations, at different levels of abstraction, in an attempt to visualize the inherently invisible entity.

The inherent invisibility of software makes it difficult to observe subtle defects. A single misstated symbol in a million-line software program can create a catastrophic system failure that may only occur under specific conditions that arise after thousands of heretofore-successful program operations.

Another unfortunate result is that software under development is often reported to be "almost complete" for long periods of time with no objective evidence to support or refute the claim; this is the well-known "90% complete syndrome" of software projects. Many software projects have been cancelled after large investments of effort, time, and money because no one could objectively determine the status of the software or provide a credible estimate of a completion date or the cost to complete the software.

There are well-known techniques that can be used to ameliorate the 90% complete syndrome of software development[3] but, unfortunately, some software organizations do not apply these techniques in a systematic manner.

Three additional factors

In addition to the four essential properties of software identified by Fred Brooks, there are three additional aspects of software engineering that should be understood by systems engineers; each results from the intangible nature of software:

1) Software engineering, to a greater degree than other engineering disciplines, involves intellect-intensive work;

2) that work is performed by closely coordinated teams; and

3) software engineering metrics and models are different in kind from the metrics and models of traditional engineering.

Certainly, every kind of engineer engages in intellect-intensive work but the lack of physical properties in software blurs the boundary between the analysis-and-design phases and the construction phase of system development. The work products of software engineers flow from their thought processes directly into the source code representation of the software that they construct at their keyboards. But, as Michael Jackson has observed, the entire description of a software system or product is usually too complex for the entire description to be written directly in a programming language, so we must prepare different descriptions at different levels of abstraction, and for different purposes[5].

Thus, systematic development processes and intermediate work products are required in software engineering, as in other engineering disciplines; however, the reasons are different. In software engineering, it is possible (although not recommended) that an acceptable software product could be developed without systematic analysis and design; the purpose of systematic development of software is to control complexity. In other engineering disciplines the purpose of systematic analysis and design is perhaps to control complexity but primarily to produce blueprints, schematics, and other plans for construction of a physical artifact.

The second additional factor to be considered is the closely coordinated teamwork required to produce software. Because software engineering is intellect-intensive, effort is the fundamental unit of estimation and control for software projects. A software project estimated to require 100 staff-months of effort might be constructed by 10 people working for 10 months but not 100 people working for one month and probably not 1 person working for 100 months; teams of individual contributors are thus required (This fact is the basis for the title of *The Mythical Man-Month* by Fred Brooks; people and time cannot be arbitrarily interchanged on a software project). The problems encountered by teams engaged in teamwork to construct software are similar to those that would be encountered in writing of a book by a team of individual contributors/collaborators, especially if those team members are geographically dispersed.

A third additional factor that distinguishes software engineering, in addition to Fred Brooks four essential properties, is the metrics and models used in software engineering. During the 20th century, great advances were made in the traditional engineering disciplines based on development of analytical models and quantitative measures. It is often the case that a physical artifact can be characterized by a few parameters such as, for example, voltage level, current flow, and heat dissipation; or mass, volume, and strength of materials. Metrics of interest can often be determined from mathematical models of the artifact in question. These metrics then guide design and fabrication of physical components and systems.

Because software has no physical properties, these kinds of models are not possible. There are no mathematical models that can accurately predict the safety, security, or reliability of software with the degree of precision that is possible for physical artifacts. This is not to imply that there are no models in software engineering (such as queuing models for software throughput) but the models and metrics in software engineering are fewer and different in nature than the models and metrics of traditional engineering.

Software development processes

The intangible nature of software may leave the impression that software is infinitely malleable and can be formed into any desired configuration without systematic development processes. However, software like other artifacts of engineering is (or should be) developed by application of systematic processes of analysis, design, construction, integration, verification, and validation. As stated above, control of complexity is the primary reason for systematic development of software. Efficient and effective development of software are also important reasons to use systematic development processes; like all engineers, software engineers should seek solutions that are timely and economical.

The intangible nature of software allows iteration among and interleaving of the phases of the development process to a much greater degree than is possible for physical artifacts. Iterative processes for software development include the incremental, evolutionary, agile, and spiral approaches[3].

While it is true that incremental and iterative processes can be, and are, used to develop systems composed of physical artifacts, it is also true that the nature of systems engineering (i.e., specifying diverse components and allocating requirements to them, accomplishing system design, and developing plans and enacting integration, verification, and validation) involves functional decomposition and linear development processes to a greater degree than in modern software engineering practice.

Smooth integration of the development processes used in systems engineering and software engineering is a continuing and ongoing challenge.

Risk Management

Risk management is, or should be, the concern of all engineers; both during initial analysis and design and on a continuing basis during initial system development and subsequent modifications. All engineering disciplines must deal with process-based risk factors, such as schedule, budget, cost, and personnel; however the technical risk factors for software engineering, because of the nature of software, are different in kind from the technical risk factors for traditional engineering disciplines.

Technical risk factors in software engineering typically involve issues such as adequacy of the computing resources (memory and processing speed of the hardware processors, distributed processing, bandwidth, etc); adequacy of the development environment (operating systems, programming languages, database tools); familiarity of the developers with the development tools and the application domain; interfaces to the hardware, software, and human operational environments, and achievement of non-functional requirements (e.g., safety, security, reliability, adaptability).

Systems engineers and other traditional engineers are also concerned with the risk factors that may be encountered in achieving the non-functional requirements for their systems. Other risk factors that may be encountered in traditional engineering projects include performance shortfalls of manufactured components (e.g., excessive power consumption, overheating, inadequate strength of materials), delays in fabrication of components, mismatches in the physical interface connections between physical components, and backlogged orders for needed components.

Pedagogical issues

The major topics to be covered in teaching software engineering concepts to systems engineers include:

- Differences in use of shared terminology
- System engineering techniques used by software engineers
- Software engineering techniques used by systems engineers
- The intangible and malleable nature of software
- The four essential properties of software
- The three additional factors
- Software development processes
- Risk management of software projects

The important concepts of software engineering can be conveyed to systems engineering students though the use of analogies, case studies, and examples. For instance, the malleability and resulting ease of changing software can be illustrated using a flowchart or pseudo-code to show, for example, changing a bubble sort routine to sort in descending order rather than ascending order. The ease of making a mistake that sorts data in the wrong order can also be illustrated.

The difficulty of changing software can be illustrated by an example that creates undesired side effects of making a change. Design techniques such as encapsulation and information hiding to control ripple effects can be illustrated. The complexity of software can be illustrated using a pseudo-code example of a recursive factorial algorithm or a quicksort algorithm. A simple, inductive proof of the recursive factorial algorithm can be shown as well as the equivalent iterative version of the factorial algorithm and the use of loop invariants. The strength and limitations of formal methods in software engineering could be presented.

The conformity required of software can be illustrated by examples such as the Mars Orbiter crash caused by a mismatch of parameters in a software interface[6] or a similar cause of the crash of the Ariane 5 rocket[7]. Invisibility and safety issues for software can be illustrated by using the race condition that, in part, caused the Therac-25 machine to overdose radiation patients[8]. These case studies also illustrate failures in systems engineering.

Other classic examples abound that illustrate the nature of software, software engineering practices, and the interactions of software engineering and systems engineering.

There are several ways in which the concepts of software engineering can be taught to systems engineering students, including:

- Software engineering concepts introduced in a freshman introduction-to-engineering course with reinforcement of the concepts in later courses

- A software engineering course tailored to the needs of systems engineering students; i.e., not a programming course

- A capstone course that focuses on software systems engineering

Many engineering schools have a first course in engineering that, in part, surveys the various fields of engineering. Software engineering could be included among the engineering fields surveyed. Later courses could include case studies in which software contributed to the success or failure of a complex system with exploration of the underlying software engineering issues (both positive and negative).

Ideally, a software engineering course (not a programming course) could be included in a systems engineering curriculum. Admittedly, this may not be feasible because it would require finding room for the course in already crowded curricula, and it might not be cost-effective to offer a specialized course in software engineering for systems engineers. However, the course might be offered to students in other engineering disciplines to amortize the course overhead among more students.

Also, capstone courses could be tailored to focus on software systems engineering, either as special offerings or as a requirement for all students.

Finally, it should be observed that many graduate students in systems engineering and practicing systems engineers could benefit from learning the material presented in this paper.

Bibliography

1. Section 1.5 of *The System Engineering Body of Knowledge, v0.25* (note: v0.25 of SEBoK is not publicly available. This reference will be changed prior to submission of the final version of the paper when v0.5 of SEBoK will be publically accessible.)

2. *A Guide to the Project Management Body of Knowledge (SPBOK), Fourth Edition*, Project Management Institute, 2008.

3. *Managing and Leading Software Projects* by Richard E. (Dick) Fairley; published by Wiley, 2009.

4. *The Mythical Man-Month* by Fred Brooks; published by Addison-Wesley, 1995.

5. "Descriptions in Software Development," by Michael Jackson in *Lecture Notes in Computer Science, Springer Verlag GmbH, Volume 2460, 2002.*

6. *Mars Climate Orbiter Mishap, Investigation Board Phase I Report*, NASA, November, 1999. http://sunnyday.mit.edu/accidents/MCO_report.pdf

7. *Ariane 5 Flight 501 Failure, Report by the Inquiry Board*, http://www.di.unito.it/~damiani/ariane5rep.html

8. *Therac-25,* http://en.wikipedia.org/wiki/Therac-25