The Journal of Systems and Software 125 (2017) 354–364

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Software Systems Engineering programmes a capability approach[☆]

Carl Landwehr[a], Jochen Ludewig[b], Robert Meersman[c], David Lorge Parnas[d,*], Peretz Shoval[e], Yair Wand[f], David Weiss[g], Elaine Weyuker[h]

[a] *Cyber Security Policy and Research Institute, George Washington University, Washington, DC, USA*
[b] *Institut für Software, Universität Stuttgart, Stuttgart, Germany*
[c] *Institut für Informationssysteme und Computer Medien (IICM), Fakultät für Informatik, TU Graz, Graz, Austria*
[d] *Middle Road Software, Ottawa, Ontario, Canada*
[e] *Ben-Gurion University, Be'er-Sheva, Israel*
[f] *Sauder School of Business, University of British Columbia, Vancouver, BC, Canada*
[g] *Iowa State University, Ames Iowa, USA*
[h] *Mälardalen University, Västerås, Sweden and University of Central Florida, Orlando, FL USA*

A R T I C L E I N F O

Article history:
Received 24 May 2016
Revised 22 November 2016
Accepted 19 December 2016
Available online 23 December 2016

Keywords:
Engineering
Education
Software education
Information systems
Software design
Software development
Software documentation

A B S T R A C T

This paper discusses third-level educational programmes that are intended to prepare their graduates for a career building systems in which software plays a major role. Such programmes are modelled on traditional Engineering programmes but have been tailored to applications that depend heavily on software. Rather than describe knowledge that should be taught, we describe capabilities that students should acquire in these programmes. The paper begins with some historical observations about the software development field.

1. Background

Many universities have created educational programmes to teach the development of software intensive systems. There is a great deal of variation among these programmes and a number of programme names are used. In this paper, we use the term "*Software Systems Engineering" (SSE) to* refer to such programmes. Some types of SSE programmes are discussed in more detail in Section 5 of this paper to illustrate what we mean by Software Systems Engineering.

There have been many efforts to define *bodies of knowledge* for computing disciplines. A list of some of these efforts can be found in The Joint Task Force for Computing Curricula (2005). Some (e.g., Parnas, 1998; Lutz et al., 2014; Ardis et al., 2015) describe programmes that have been developed by individual institutions. Others, (e.g., Computing Curricula, 2005), compare the bodies of knowledge associated with various computing disciplines. In Glass et al. (2004), there is a comparison of computing disciplines based on the research areas associated with each. The SE2004 report (Lethbridge et al., 2006), (and its updated version SE 2014 (Ardis et al., 2015)), propose knowledge that should be taught in undergraduate software oriented programs. They also provide sample courses and curriculum patterns.

This paper takes a complementary approach. Noting that:

- Science programmes present an organized body of knowledge and teach students how to verify and extend that knowledge.
- Engineering programmes present an organized body of knowledge and teach students how to apply that knowledge when developing products.

Instead of discussing the knowledge that would be conveyed to students during their education, this paper focusses on things that a software developer must be able to do while developing and maintaining a product. Like Lethbridge et al. (2006), and Ardis et al. (2015), this paper discusses a set of Engineering programmes in which software development plays a central role; un-

☆ Work on this paper began while the authors served on a committee advising Israel's Council of Higher Education that was chaired by David Parnas. The opinions presented in this paper are the personal opinions of the authors. All authors made substantive contributions to the paper; they are listed in alphabetical order. David Parnas is the corresponding author.

* Corresponding author.
E-mail address: parnas@mcmaster.ca (D.L. Parnas).

http://dx.doi.org/10.1016/j.jss.2016.12.016
0164-1212/© 2016 Elsevier Inc. All rights reserved.

like Lethbridge et al. (2006), and Ardis et al. (2015), it does not prescribe courses or curricula. Rather than describing knowledge or research areas, we propose a *body of capabilities*. Many different curricula could help students to acquire the capabilities described in this paper.

Because software is a rapidly changing field, we expect that the associated "body of knowledge" will continue to grow quickly and curricula will need to be revised frequently. In contrast, the capabilities discussed in this paper are fundamental. We base them on observations that were made when the profession of software development was first identified (Brooks, 1995; Buxton and Randell, 1969; Naur and Randell, 1968). They were needed then, they are needed now, and we expect them to be needed in the far future.

We do not believe that the capability approach is a replacement for "Body of Knowledge" or Curriculum proposals. We believe that looking at capabilities as this paper does, provides a perspective that will help institutions to develop, compare, and update curricula.

- Section 2 of this paper reviews discussions that took place when the term "Software Engineering" and similar terms were first introduced.
- Section 3 discusses some capabilities that Software Systems Engineers need.
- Section 4 discusses the role of projects in Software Systems Engineering education.
- Section 5 describes a few of the many distinct disciplines that fall under the rubric of Software Systems Engineering.
- Section 6 discusses how to use this paper when designing or revising a curriculum.
- An appendix provides a more detailed discussion of the most important learning outcomes for Information Systems Engineering.

2. Searching for a definition of "Software systems engineering"

In the 1960s, some computer scientists began to use the phrase "Software Engineering"[1] without providing a clear definition. They expressed the hope that software developers would learn to construct their products with the discipline and professionalism associated with professional engineers (Buxton and Randell, 1969; Naur and Randell, 1968).

When the term "Software Engineering" was first introduced, many asked, "How is that different from programming?" More recently, when post-secondary "Software Engineering" programmes were introduced, some asked, "How is that different from Computer Science?" Some who asked these questions questioned the need for a new term; others wanted to know what, beyond programming and computer science, would be taught to students of "software engineering". In the discussion that followed, two simple, but consequential, answers emerged. Although both definitions are old, they have withstood the test of time, are consistent with current usage, and remain relevant today

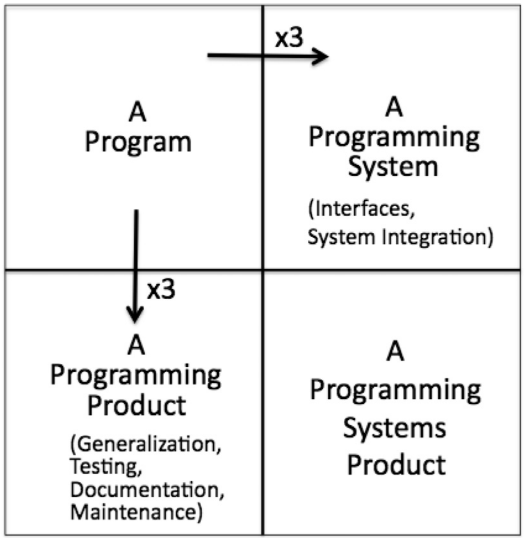
2.1. Brian Randell's answer

One of the best answers to the question was provided by Prof. Brian Randell, one of the organizers of the first two international Software Engineering conferences and co-author of two frequently referenced reports on those meetings (Buxton and Randell, 1969; Naur and Randell, 1968). In private discussions, he described Software Engineering as "multi-*person* development of multi-*version* programs". This pithy phrase implies everything that differentiates professional software engineering from programming. Software engineers must be able to work in teams to produce programs that will be used, and revised, by people other than the original developers. Although performing that job requires programming skills, many other capabilities are required as well.

[1] Historically, the term "Software Engineering" was used. However, we believe that what is said in this section applies to all Software Systems Engineering disciplines.

2.2. Fred Brooks' answer

The diagram below appears In Fred Brooks' classic book, "The Mythical Man-Month" (Brooks, 1995). The vertical dimension denotes "productizing" and the horizontal one "integration".


x3
A Program
A Programming System
(Interfaces, System Integration)
x3
A Programming Product
(Generalization, Testing, Documentation, Maintenance)
A Programming Systems Product

Fred Brooks' explanation of why software engineering is more than programming.[2]

- By testing, documenting, and preparing a program for use and maintenance by other people, one transforms that program to a "programming product".
- By integrating a program with other, separately written, programs, one moves from a program to what Brooks called "a programming system".
- Doing both of these results in a "programming systems product". Going from a program to a programming systems product results in a massive increase in cost and effort.

Brooks' formulation, like Randell's, makes it clear that there is much more than programming skill required of a software engineer. Software engineers must master programming, but they must also be able to integrate separately written programs and "productize" the result.

3. What should Software Systems Engineers be prepared to do?

The decision to create Software Systems Engineering programmes that are distinct from "Computer Science" programmes makes these old questions relevant today. We have to ask how Software Systems Engineering programmes should differ from Computer Science Programmes and what criteria should be applied when evaluating them.

[2] Figure redrawn from (Brooks, 1995). The "x3" annotation, denotes a 3-fold increase in effort.

This section lists some activities that are implied by the answers offered by both Randell and Brooks. We believe that Software Systems Engineering programmes should teach the fundamental principles and procedures that will help their graduates to perform these tasks well. Rather than prescribe the knowledge content of a Software Systems Engineering programme, we describe a core set of capabilities that should be acquired in those programmes.

This paper deliberately avoids naming any particular techniques, technologies and "methodologies". For example, although we suggest that graduates should have learned how to create and use models, we do not mention any modelling languages, methods, or tools. Further, the choice of how, and when, to teach modelling is left to the individual institution.

We have five reasons for not naming specific tools or languages in this paper:

- Software development technologies change rapidly and quickly become outdated. A post-secondary education should prepare students to learn new technologies as they are developed; graduates should also be prepared to learn an old technology if asked to work on a product that was originally developed using tools that are no longer in common use.
- Many current technologies are commercial products whose usage and effectiveness are exaggerated by advocates who often act more like sales representatives than professional scientists or engineers.
- Some departments will have favourite tools or languages that fit their view of how things should be done; they should be allowed to use those tools in their teaching.
- The development of a new method or tool should not obsolete the education of an engineer who was educated before it appeared.
- It is more important that graduates of a Software Systems Engineering programme are able to use the available evidence to evaluate new methods and technologies, and then choose the approach that is best for their present project, than that they have learned about any particular tool or notation.

However, one cannot teach engineering methods and concepts without giving students a chance to apply those approaches while they can benefit from guidance by their instructors. Consequently, during their education, students will have to learn to use some of the current tools. To help students to distinguish between fundamental principles and current technology, it is often useful to separate the teaching into lectures and laboratory sessions so that

- the lectures teach fundamental concepts and principles, and
- the laboratory sessions provide experience applying the lecture material using reasonably current tools and notations.

The authors are quite aware of current developments in Software Systems Engineering programmes and the explosion of topics and terminology that has resulted from the increased level of activity in our field. It would be impossible for us to discuss all of the issues and concepts that have been discussed in the academic and industrial literature. We have chosen instead to focus on issues and capabilities that we consider to be fundamental and hence a kernel. Each institution can build on this kernel in its own way.

We expect that the notations and tools that are popular today will be replaced by newer tools and approaches. In contrast, we believe that the capabilities discussed below will remain important throughout a graduate's career.

The following sections discuss specific capabilities.

3.1. Communicate precisely between developers and stakeholders

For any product development to be successful, the needs and preferences of the stakeholders (e.g., purchasers, clients, investors, present and future users) must be communicated to the developers. It is best if these requirements can be determined and documented early in the development process.

Understanding user needs before a system is available is very difficult. Often the clients do not know what is required; even when they know, user representatives may not be able to communicate those requirements clearly. Consequently, their statement of requirements will change during the project and will continue to change throughout the period in which the system is used. Developers must be prepared to respond to the changes that (inevitably) occur both during system development and after deployment. Nonetheless, we believe that there are many advantages to having the best possible description of the client's requirements before code writing begins. Of course, the developers must be prepared to revise that description to keep it consistent with the clients' needs as the product develops and understanding grows.

As in other engineering fields, determining and documenting requirements requires extensive interaction between potential users (or their representatives) and representatives of the developers; it may require modelling (see Section 3.8 of this paper) and building prototypes (simulators or mockups) that potential clients can use and critique. The goal of these efforts by developers and stakeholders should be to produce requirements documentation that is precise, readable, and demonstrably formally complete[3].

Often, the stakeholders' representatives will not have been trained in requirements specification. In such situations, the Software Systems Engineers will have to take the lead by producing a draft statement of requirements that can be reviewed, corrected, and (eventually) accepted by the purchasers or by user representatives.

After the development is complete, the user-visible properties of the product must be communicated to the end-users so that they can make effective use of the system. This, too, is part of the responsibility of a Software Systems Engineer. While professional technical writers can help in the preparation of user documents, only the developers have the requisite system knowledge.

In short, Software Systems Engineers should learn how to elicit and document requirements and how to describe the visible behaviour of a completed product; they should understand the importance of keeping these documents up to date as user preferences and system behaviour evolve; they should know how to organize requirements documentation so that updating is relatively easy.

3.2. Communicate precisely among developers

When groups cooperate on a project of any type, the ability to communicate precisely about many things (e.g., goals, schedule, resources, standards, and interfaces) is essential to their success.

Most software development projects are organized as a set of tasks; the ultimate goal of most of those tasks is to design, develop, test and deploy groups of programs that we call *modules*; each module becomes the responsibility of a group of developers[4] or an individual developer. Those developers must know both:

[3] By "formally complete" we mean only that the documentation characterizes acceptable behaviour in all possible situations. Formal completeness should not be confused with correctness; it does not guarantee that the behaviour identified as acceptable by the document is actually suitable for the intended use. It is possible for a Software Systems Engineer to check a document for formal completeness; one needs stakeholders to check for correctness.

[4] Modules may comprise several smaller modules (submodules).

- how their module is required to behave, and
- the behaviour that they can expect of other modules.

If the required behaviour is not clearly and completely specified, the product will be plagued by missing capabilities, redundant capabilities, and incompatible programs. It is a basic property of digital technology that very small misunderstandings can lead to major failures. Describing module interfaces precisely has proven particularly difficult for software projects because the descriptions must specify the behaviour of the modules over time. The tools used in traditional Engineering to do this (e.g. differential equations) are not helpful with software because the functions that characterize software behaviour are not differentiable.

Understanding the nature of an interface is the key to successful inter-developer communication. Many problems in software development arise because developers have an overly simple view of what constitutes an interface. Below are examples of distinctions that are often overlooked in CS courses. If graduates understand these distinctions, *and learn how to apply them*, they will be better software developers.

- The assumptions that the developers of a module, A, *are allowed* to make about another module, B, constitute B's *specified interface* to A.
- The assumptions that the developers of a module, A, *actually* make about another module, B, constitute B's *actual interface* to A.
- B's actual or specified interface to A may be different from B's actual or specified interface to other modules.
- A's actual or specified interface to B is not generally the same as B's actual or specified interface to A.
- In practice, an actual interface may differ from the corresponding specified interface.
  - If B's actual interface to A includes assumptions that are not implied by the specified interface, A may fail even if B functions as described in the specified interface.
  - If B's actual interface to A is a proper subset of the assumptions allowed by the specified interface, A won't fail as a result but may be less efficient than it could be.

Because even small deviations from an intended interface can cause major changes in software behaviour, interface documents should be unambiguous and cover all possible usage sequences (i.e., be formally complete).

In short, Software Systems Engineers should learn how to design module interfaces and how to read and write module interface documentation.

3.3. Design human-computer interfaces

Most Software Systems Engineers will be working on systems that present information to, and receive information from, people. When designing human-computer interfaces, Software Systems Engineers should learn to consider both the nature of the information that will be exchanged and the capabilities and habits of the intended users.

Software Systems Engineers should also learn to distinguish between ease-of-learning and ease-of-use so that they can design interfaces for both beginning and experienced users.

Designing human-computer interfaces is a multidisciplinary subject. It combines knowledge from fields such as psychology, engineering biomechanics, industrial design, and graphic design, with an understanding of software design and the characteristics of interface devices. There are software packages designed to ease the construction of user interface software; a Software Systems Engineer should understand what such products can do and know how to choose the one that is best for their project.

In short, Software Systems Engineers should know how to design human-computer interfaces that are easy to use and improve users' productivity.

3.4. Design and maintain multi-version software

Successful software products evolve but the older versions don't necessarily disappear; often, the older versions must remain in the development organization's product line. Knowing how to design and maintain software product lines is essential for Software Systems Engineers.

Designing products for ease of change requires approaches that often seem counterintuitive to programmers. These approaches sometimes lengthen the development time for the first version in order to reduce both the time required to develop later versions and the cost of maintaining several versions simultaneously.

There are several approaches to software design that make a product easier to change. All of them require the developers to take the time to think about possible changes during the initial design. Some of the approaches to making software easier to change that can be taught to Software Systems Engineering students are discussed in more detail below.

3.4.1. Identify and separate changeable concerns

Software revisions are easier if the software is designed so that the most likely changes are localized in modules that can be revised without affecting other modules. Software Systems Engineering students should learn how to:

- study the product's requirements to identify the aspects that are most likely to change,
- study the support system and hardware to identify aspects that are likely to change,
- study the software design decisions (algorithms, module interfaces, and data structures) to identify the ones that are most likely to require change,
- organize software as a set of modules that can be developed, and changed, independently because each module has complete responsibility for a changeable aspect of the system, and
- design module interfaces that abstract from the changeable aspects of the module so that the interface is not likely to change even if the module implementation is revised.

Experience has shown that, even if unanticipated changes are eventually required, software designed for ease of change is easier to maintain than software designed without concern for future changes.

In short, Software Systems Engineers should learn to prepare for change by thinking about what is likely to change and encapsulating the most changeable aspects of their work in well-specified modules.

3.4.2. Document to ease revision

When change is needed, the code may be revised by people who did not write it. Even if the original programmers make the revisions, they may no longer remember the details and the reasons for the decisions that they made. Consequently, developers should leave a detailed and precise design description for the maintainers. Software Systems Engineering students must learn how to:

- produce detailed design documentation that can serve as a "knowledge base" for future developers,
- organize documentation so that information is easy to find,
- organize the documentation so that information that is likely to change is not repeated, and

- maintain all artifacts (both code and documents) in an accurate and up-to-date state so that the documents can serve as a reliable source of information for future developers.

In short, Software Systems Engineers should learn how to produce and maintain orderly design records to help future maintainers.

3.4.3. Use parameterization

A powerful tool for making software easier to change (see 3.4.1) is parameterization. Software Systems Engineers should learn how to spot opportunities to parameterize their designs and use the parameters in both documents and code. This will make their design more generic and avoid distributing constants that are likely to change throughout the product.

3.4.4. Design software that can be moved to many platforms

With the wide variety of platforms available today, it is important that software developers be able to develop and maintain products that can be used on several of them. Designing software to be portable is a special case of designing for change (3.4.1) but there are special issues that a software developer must be able to resolve.

Porting software to new platforms can lead to conflicting requirements and difficult design decisions for the developer. For example:

- Users of a platform may prefer an application to have the same “look and feel” as other applications on that platform.
- Users of a software product may want it to have the same “look and feel” on all of their platforms.
- One platform may provide services that make it easy to offer a capability that would be difficult to offer on some other platforms.
- The mechanisms available for transferring data between applications may differ between platforms.
- Error handling conventions can differ between platforms.

Industry has had mixed success in solving these problems. Often product lines have features that are available on some of the platforms but not available on others. This can confuse and annoy users.

In short, Software Systems Engineers should be aware of the problems of building multi-platform products and understand how to organize software so that the platform-specific modules are clearly identified.

3.4.5. Design software that is easily extended or contracted

Users have come to expect software products that are easily extended when new capabilities are needed. They would like the extended product to be compatible with the previous version so that:

- they can continue to use the old capabilities without changing their habits or updating other programs, and
- the extended program can work with data created by the previous version.

In other situations, there may be a need for a product with reduced capability. For example, a software company may want to offer a free version to attract consumer interest or a version that will run with limited computational resources.

If some products are extensions of others, the cost of maintaining a software product line will be reduced whenever a change can be effected by revising a shared module.

Extending and contracting software has proven unexpectedly hard for industry to achieve; often, the “extended” versions are not true supersets of the smaller ones and the various versions are not compatible.

In short, Software Systems Engineers should learn how to design software so that it can more easily be extended or contracted and data can be reliably transferred between versions.

3.4.6. Design and maintain products that will be offered in many versions

Software development organizations must often maintain many versions of their software products. It is important that students learn how to design software to make this as easy as possible. The fact that many versions of a product will be offered should be reflected in the structure of the products and their interfaces.

Multi-version products comprise a large set of components[5] and several versions of each component may exist. It can be very difficult to make sure that a particular instance of the product contains the right version of each of its components. Making sure that all products have the correct versions of their components is often called configuration management.

There are many configuration management tools available to organize the versions of components and assemble working systems. Software Systems Engineers should be able to compare these tools, understand what they can, and cannot, do, and pick the right tool for their project.

In short, students should be taught the basic principles of software configuration management and provided with an opportunity to apply those principles using at least one of the tools.

3.5. Design software for reuse

Software reuse is like good parenting; everyone is in favour of it, but it is often very difficult. Unless the software was designed and documented with future reuse in mind, it may be more difficult to reuse a module than to develop a new one. Even when an organization has code that could be reused in a new project, the developers of that product may not know of its existence or capabilities. Whenever software is being reused, it must be thoroughly tested in its new environment.

Program code is not the only software artifact that developers should reuse. Development organizations could also benefit by reusing documents, and test suites. This too is more easily said than done. Small changes between versions may subtly invalidate previously issued documentation and tests. Older documents may ignore new features or assume the presence of a feature that has been replaced. If a characteristic of the older versions has been mentioned in several parts of the documentation, those revising the document may produce an inconsistent document by changing some occurrences but not others. Test suites can be inadequate for newly added features or can falsely indicate a failure because of a deliberate change. It takes careful structuring of documentation and test suites to make reuse easier.

In short, Software Systems Engineers should learn how to design and document software in ways that make the code and other artifacts easier to reuse; they should also learn how to organize repositories for the potentially reusable artifacts so that reusable assets can more easily be found when needed.

3.6. Ensure that software products meet quality standards

The fact that software is written for use by people who did not develop it and do not understand its structure, adds to the responsibilities of those who develop the software. When we write a program for our own use, we are well equipped to detect, understand,

[5] We use “*component*” to denote the replaceable parts of software as distributed. For example, an update replaces old versions of one or more components with new versions of those components. Components may include programs from several modules and programs from a module may be included in several components.

and deal with any features that are difficult to use or failures that occur during use; most other users will not have that capability. Consequently, usability and reliability are more important for software products, than for programs intended for the programmers' own use.

Software Systems Engineers must understand that quality assurance is their professional obligation and that they should refuse to release products that have not been shown to be fit for their intended use. They should know how to follow processes that improve software quality and be proficient in methods that help to assure that the quality of a product is acceptable such as:

- reviewing software structure for conformance to accepted design principles,
- reviewing documentation for completeness, accuracy, and usability,
- testing programs — to find and eliminate faults, as well as to estimate reliability[6],
- "divide and conquer" inspection of large programming systems, and
- formal verification of critical small programs.

In short, Software Systems Engineers should be taught that they are responsible for the quality of the products that they release, must know how to apply basic methods of quality assurance, know how to design software to make inspection and testing easier, and know how to inspect and test software products.

3.7. Develop secure software

Today's software is used over networks that can be used to attack user's systems. Even software that is not connected to a network is often used by many users; some of those users may try to interfere with others. Even single-user software can, if it has not been carefully designed, be tricked into behaving in unacceptable ways and inflicting damage on its users.

A Software Systems Engineer should know how to design software that does not allow one user to either interfere with, or gain access to, the data of others. They should also know how to make sure that software is robust and can handle maliciously crafted (or erroneous) inputs properly.

Software Systems Engineers must understand that security must be a design concern from the start; many years of research and experience have made it clear that security is not achievable if it is an afterthought.

In short, Software Systems Engineers should understand the basic principles of designing programming systems that are intrinsically secure.

3.8. Create and use models in system development

The creation and use of models (physical, mathematical, and diagrammatic) plays an essential role in all engineering disciplines. A model is a simplified version or description of a product that is intended to be used for predicting the behaviour of the actual product. Great care must go into creating the model to make sure that it is suitable for its intended purpose. Because of the inaccuracies in models, even greater care is required when interpreting the results of a model-based analysis.

Because most models are simpler than the product, they do not constitute an accurate description of the product and cannot replace detailed documentation. Conversely, detailed documentation is not usually well suited for use in modelling.

[6] Reliability estimates can be used when deciding whether or not a new product is ready for release.

In short, modelling is quite different from programming and documentation; Software Systems Engineers should be able to create and analyze a variety of models and use modelling results to improve the design of the software systems that they develop.

3.9. Specify, predict, analyze and evaluate performance

Because today's software systems integrate the work of many people and are often being used by many people at the same time, resource utilization can be hard to predict and the response time of a computer system is often unsatisfactory. Even systems that compute correct values every time that they are used, will be considered unsatisfactory if they are too slow or require too much memory.

Software Systems Engineers are responsible for insuring that their products perform adequately. They must be able to specify the required performance of a system. Specification of performance requirements requires a characterization of the expected load on the system as well as anticipating the expectations of users. Prediction of the performance of a proposed design may require making models of both the system and its environment and the application of queueing theory and simulation techniques.

If a system is found to have inadequate performance, the Software Systems Engineer must be able to analyze the hardware and software to identify the causes of the problems. When a system is deemed complete by its manufacturer, Software Systems Engineers who work for the customers will need to carry out performance tests before accepting it.

In short, producing systems that perform adequately, and evaluating the performance of such systems, requires Software Systems Engineers to be competent in a variety of areas, many of which, such as queueing theory, were developed outside of Computer Science.

3.10. Be disciplined in development and maintenance

"The phrase 'software engineering' was deliberately chosen to be provocative; it suggested that software manufacture must be based on the types of theoretical foundations and practical disciplines that characterize the established branches of engineering" (Naur and Randell, 1968).

If we observe the way that engineers work in fields such as Civil Engineering, we see that they are required to follow relatively rigid processes when designing, documenting, and checking their products. These procedures often include measurements that must be made, forms that must be completed, mathematical analyses that must be performed, and standards that must be satisfied. Discipline is also required when repairing an engineering product. Engineers are not born with disciplined work habits; they have to be taught.

Analogous procedures need to be taught to Software Systems Engineers. Students should be given numerous opportunities (in the form of projects and exercises) to practice the procedures that they have been taught so that they can gain a deeper understanding of the principles behind procedures and develop good habits.

Software maintenance requires at least as much discipline as the original development. Software Systems Engineers also need to be taught how to modify other people's programs in ways that keep them consistent with the original structure of the software. When changes do not conform to the original design concepts, software starts to age and becomes hard to maintain.

Numerous tools intended to help the development team to work in a careful and disciplined way are on the market. A Software Systems Engineer must understand what these tools can (and cannot) do and know how to choose and use the best tool for a given project.

To improve the discipline of their software development teams, many companies specify a development process, i.e., a sequence of steps that their development teams must follow and a set of work products that must be produced.

Some companies choose a process that has been developed and promulgated by external experts. Those experts may be either academics or commercial consulting firms; many of their processes are publicized and supported by books and articles. Other companies invest time and effort to develop their own process and produce internal documents that describe it[7]. Even when developers say that they have followed a specific process, discussions may reveal that they have actually deviated from that process by skipping steps, modifying some work-products, or adding extra work products.

Many types of processes (e.g., continuous integration, agile development and test-driven development) have been proposed but none is appropriate for every situation. Software Systems Engineers need to be familiar with several processes, understand their intended purpose, and know the strengths and weaknesses of each.

In short, Software Systems Engineers should be taught to work with care and discipline both when they are developing a new product and when they are modifying an old one.

3.11. Use metrics in system development

Because of the complexity of engineering products, they are often evaluated, explained, and sold to others on the basis of "figures of merit" or "metrics". Many metrics have been proposed for software; experience has shown that many of these metrics are misleading in that a "better" value does not always mean a better product. Software Systems Engineers should learn what metrics are useful and the main considerations in choosing, and using, metrics.

In short, Software Systems Engineers should learn about software metrics and know how to use them with caution.

3.12. Manage complex projects

Building and installing software requires coordinating the work of developers, marketers, clients, and users. Coordinating the work of many people to produce a new system is a complex task that requires:

- planning and scheduling,
- estimation of cost, time and effort needed for a task,
- progress measurement,
- problem tracking,
- risk management,
- configuration management,
- resource control,
- task assignment,
- forming teams and other organizations (e.g. matrix organizations),
- choosing and using project management tools,
- leadership.

There are two contrasting views about the relation between project management and Software Systems Engineering:

- Project management is an important problem for all engineering disciplines. Project management is often taught by both Management and Engineering faculties; much of what is taught is independent of the nature of the product. Consequently, one can view project management as a discipline that is outside of Software Systems Engineering. Some institutions include a semester course on project management in their general engineering programme. Others offer project management as an extension to an engineering degree (extra year) or offer engineering project management at the Masters level for students who already have an engineering degree.
- Experience suggests that project management is more difficult in software projects than it is in other engineering projects of a similar size. Problems arise in software projects that are not common in other disciplines. Consequently, some believe that software project management should be an integral part of a Software Systems Engineering programme.

Several reasons have been advanced for treating project management differently from the way that it is treated in other engineering disciplines.

- Software project management is made more difficult by the fact that plans and decisions often have to be made using incomplete and insufficient information.
  - The lack of complete information is most noticeable for the important decisions that are made early in the development process when requirements are not well understood and are poorly documented.
  - Requirements that were stated early in the project are often revised later.
  - During the development, internal interfaces are often incompletely documented; as a result, much time is spent coordinating groups and adjusting the interface between modules to make those modules compatible.
- Many software products are new and innovative; consequently, estimation of cost and benefit is more difficult.
- Often, managers and developers lack experience with a new approach and cannot accurately predict how well their approach will work.
- The structure (aka architecture) of the software is often such that changes in one module may seriously affect the completion time of others.
- For software products, there is often no clear end to the development period; development commonly continues after the release of the first version.
- Because a software development organization must often maintain many versions of its software products, those products are composed of a large set of components; further, several variations of each of those components may exist. It can be very difficult to make sure that a particular version of the product contains the right version of each component.

In short, whether one considers project management as an integral constituent of Software Systems Engineering, or views it as a separate topic, every Software Systems Engineer should understand the basics of project management. There is a great deal of substantive material that can be taught in an academic programme. It can either be taught in courses shared with other Engineering Disciplines or integrated into the Software Systems Engineering programme.

4. The role of projects in Software Systems Engineering education

Professional educational programmes (e.g., engineering, medicine, and law), are intended to teach "how to" as well as "about". In particular, while engineering programmes teach a great deal of science and mathematics, they have an additional obligation; they must make sure that their students know how to apply that material when developing products.

[7] They do this because they believe that their company has characteristics that make the externally developed processes a poor fit for them.

Because the application of scientific knowledge is so important in engineering, laboratory exercises and small projects should be used throughout the curriculum so that students can experience how the theory that they hear about in lectures can be applied in practice. The importance of these projects is often underestimated. Projects should not be random "jobs" that some "customer" needs; they should be carefully designed to give students the best possible learning experience. Projects should be treated as major constituents of a Software Systems Engineering programme; all work should be carefully evaluated, and detailed feedback should be provided to students. Instructors must be given sufficient time and resources to do this.

Many engineering programmes use a final-year project (sometimes called a "capstone project") to provide students with an opportunity to consolidate what they have been taught by using that knowledge while they can still get guidance and supervision from their teachers. Such projects allow them to see how the many facts, concepts and procedures that they have learned in earlier years can be combined to form a coherent process for producing products. In a well designed educational project, all students will play a few roles[8], which will give them a broad view of the product production process.

Because "multi-person" is one of the defining characteristics of Software Systems Engineering, it is important that, in the projects in a Software Systems Engineering program, the student experiences:

- organizing a multi-person project as a set of fully specified individual tasks,
- completing one or more of those individual tasks,
- integrating separately developed modules to produce a high quality product that can be used by others, and
- experiencing the reaction of others to using their product.

In the capstone project, every team member should participate in discussions about project management and interfaces.

If the project is properly supervised, the students will have a valuable learning experience, their completed individual projects can be used by the faculty to evaluate their ability (i.e., give them a grade), and the graduates will produce a "portfolio" of products (both code and documents) that can be shown and demonstrated to prospective employers.

5. Variants of Software Systems Engineering

Institutions that offer Software Systems Engineering programmes must choose between breadth and depth. They will have to balance teaching general software development principles against focussing on a particular class of applications. Each has both advantages and disadvantages.

- The graduates of a general Software Systems Engineering programme will be able to work in many application areas, but may lack familiarity with particular areas of knowledge that are relevant for some applications.
- Graduates of a specialized programme will be highly qualified for work in their own application area but may require extra education and effort if they switch to a different class of applications.
- Graduates of a specialized programme are usually well prepared to work with non-software professionals in that application area.
- Graduates of a general Software Systems Engineering programme may not be prepared to discuss technical details with some of their non-software colleagues.

Whatever choices an institution makes, its programme should be clearly described so that prospective students and employers can make informed choices.

Below, we illustrate the breadth of the Software Systems Engineering field by listing a few possible programmes[9]. This list is far from complete; it is intended only to illustrate the broad range of choices available to educational institutions.

5.1. Communications system software engineering (CSSE)

The world has been transformed by the availability of systems that are capable of transmitting information over long distances at high speeds. Software is at the heart of telephone switching systems, mobile telephony, the internet, broadcasting systems, local area networks, etc. Software Systems Engineers who practice in this area require knowledge of how communications systems work, communication protocols, network interfaces, and applied information theory.

5.2. Information Systems Engineering (ISE)

Modern organizations depend on the rapid availability of information about their environment, employees, business processes, ongoing operations, and customers. They need systems that can support their work by finding, filtering, and presenting the right information to an employee, manager, or customer at the right time.

Software and data bases are critical technologies in building systems that satisfy an organization's need for information. Information systems engineering programmes should equip their graduates with methods, knowledge, and skills that will enable them to develop computer-based information systems that are tailored to the organizations they serve.

Information Systems Engineers require more understanding of organizational structures, behaviour, decision making, and business processes, than other types of Software Systems Engineers.

The appendix discusses learning requirements for ISE programmes in more detail.

5.3. Mechatronics Engineering (Software intensive engineering)

Software is now replacing analog technologies in many classical engineering applications. Software is critical in the control of manufacturing systems, transportation systems, robotics, navigation systems, aircraft design, weapons systems, etc.

Software Systems Engineers who will practice in these areas may require an understanding of many topics that classical engineers study, e.g., physics, differential and integral calculus, chemistry, thermodynamics, and materials science.

5.4. Software Engineering

"Software Engineering" (SE) usually denotes a broad Software Systems Engineering programme that is intended to be application independent. SE programmes teach students about many types of software rather than specializing in a particular class of products. Because there are no application-specific requirements, it is possible to allow students to go more deeply in specific areas than would be possible in an application-specific Software Systems Engineering programme. Some possible areas for additional depth would be:

[8] Possible roles would be requirements analyst, modeller, interface designer, implementor, tester, module user, documenter, manager, etc. A student should be assigned several different roles but need not cover all possible roles.

[9] The programmes are listed in alphabetical order.

- Computer science topics such as graphics, robotics, or search algorithms,
- Software Systems Engineering topics such as quality assurance, security, interface design, or documentation,
- Mathematics topics such as graph theory, logic, queueing theory, differential equations, or statistics.

5.5. System-Software Engineering

All modern software is built using “system software” such as operating systems, device drivers, network interfaces, compilers, data base systems, and file systems. These products are often hardware dependent and have especially stringent reliability, security, speed and resource utilization requirements because all other software in the system depends on system software functioning correctly and efficiently. System-software engineers specialize in this type of product and will require specialized knowledge in concurrency, run-time error handling, resource allocation, system security, and hardware/software interfaces.

6. Curriculum design

Because there are many useful variants of Software Systems Engineering, we suggest that institutions carefully discuss which of the variants described in Section 5 are both suitable for their students and match the strengths of their faculty members. There are many useful variants and it is not desirable to have all institutions offering the same ones. Each variant should be clearly labelled and described so that potential students can make an informed choice.

Believing that there are many valid ways to teach students who want to be Software Systems Engineers, and many ways to organize the material into courses, we suggest that the capabilities listed in Section 3 of this paper be used as a checklist when designing or revising a curriculum. Institutions should look at each of the capabilities listed and ask, “How will our students learn to do that?”

Many methods have been proposed for performing each of the tasks mentioned in this paper; curriculum designers will have to decide which methods to teach. We expect the methods to evolve as the field develops.

There are also many ways to teach each method. Further, some methods can be introduced or mentioned in courses but students can best obtain experience in using them when they are on the job. Curriculum designers must think carefully about how best to use the limited time available to them.

This paper focuses on those aspects of Software Systems Engineering that differentiate it from Computer Science and programming. We have emphasized the most basic (core) capabilities. Obviously, the relevant aspects of Computer Science and programming and some current issues must also be included in these programmes. Students need both the basic capabilities we have described and the ability to “hit the ground running” when they enter industry.

Each institution will have to decide how it will help its students to obtain the necessary capabilities, how it will package the concepts and techniques into courses and projects, and how much attention each capability will get in their programmes.

7. Summary

Brian Randell, Fred Brooks, and others clearly identified the key differences between Computer Science, a research field, and Software Systems Engineering, a class of professions. Unfortunately, the (more than) four decades that have passed since then, have not seen enough progress in the professionalism and discipline of software developers. There is a great need for programmes that prepare people for a profession as a Software Systems Engineer.

It is important that programmes that are identified as Software Systems Engineering programmes help their students to acquire the capabilities discussed in Section 3 of this paper.

Finally, Software Systems Engineering programmes should never be viewed as extended Computer Science programmes or as “advanced programming” programmes. Professional software system development is much more than programming and requires many capabilities that are not usually taught in Computer Science programmes. Further, some parts of Computer Science are not essential for Software Systems Engineers; curriculum designers must exercise judgement about teaching material not included in the core that we have outlined.

Appendix. Learning requirements for Information Systems Engineering

For each of the variants of Software Systems Engineering, other than Software Engineering[10], it is necessary to formulate additional programme requirements that are specific to that (more specialized) discipline. Because several of the authors of this paper specialize in Information Systems Engineering (ISE) programs, we have prepared a list of capabilities that we consider especially important for information systems engineers.

A. Structure and manage organizations

Information systems engineers should have an understanding of organizational structures, organizational behaviour, business processes, decision making, human resource management, accounting, finance, marketing, operations and logistics management.

Without an understanding of how an organization works, possible alternative organizational structures, and the information needs of individuals in the organization, ISE graduates will find it difficult to design effective information systems.

B. Analyze the information needs of organizations

Information Systems Engineers must be able to understand an organization’s need for information technology and systems, evaluate alternative IT solutions, and determine the best approach. An important tool for analyzing an organization is “*business process modelling*”. Business process modelling helps the ISE to understand how the organization operates, and then to identify its information systems needs.

The analyst must be able to:

- create models of an organization, often in the form of graphs, in which the elements are subdivisions or steps in the organization’s business processes,
- model the behaviour of the organization’s subdivisions, or steps in its processes, using input/output relations, and
- derive the behaviour of the model.

Many types of organizational charts and associated analytical methods can be useful to information systems engineers throughout their careers. There are many tools designed to help an analyst to create and use such charts. An Information System Engineer should understand the assumptions underlying each tool and be able to select the best tool(s) for a project.

In short, system problems arise in studying both the organizations that use information systems and the information systems

[10] Software engineering is an exception because it is not specialized for a particular application domain.

themselves. Information systems engineers should learn a variety of methods for solving these problems.

C. Analyze and organize large amounts of data

Modern organizations have access to vast amounts of data; the individuals who make decisions for the organization are often unable to process the raw data to get the information that they need. Today's organizations use information systems to "mine" the available data and build a "web of data" (sometimes called a semantic web) that can provide the organization with the information that it needs to operate efficiently.

Two basic approaches to dealing with large amounts of data are filtering and aggregation. When filtering, data considered irrelevant to the person receiving the information are eliminated. Aggregating computes functions of the data items to provide summaries that give the recipients more easily used information.

The classic problem of exploiting an organization's data resources has recently become a popular research field known as "Big Data". The data can come from many sources and may not be consistent. It is important for Information systems engineers to be able to apply the work being done in this area.

In short, information systems engineers should be able to use such techniques as filtering and aggregating when designing information systems for their customers.

D. Recording the provenance of information and ascertaining its quality

The information available to organizations comes from a broad variety of sources of varying trustworthiness; consequently, some data must be treated as lower quality[11] than other data. Further, information from different sources may not have been defined or measured in the same way. Finally, information obtained in the past may no longer be valid when it is used.

In short, information systems engineers should know how to keep track of the sources of the information that they process, how to assess the quality of information in their data bases and inputs, and how to deal with information that is not completely trustworthy.

E. Analyze risk exposure and use scientific decision making methods

Many advanced organizations make decisions based on quantitative models that provide an analytical approach to decision making and problem solving. These include operations research, optimization, and mathematical methods of risk analysis. An organization's information systems may be expected to apply such methods in order to help managers to make decisions. Information systems engineers should have an understanding of quantitative, rule-based, and other decision making procedures so that they can incorporate them in the systems they build.

In short, an information systems engineer should be able to apply "management science".

F. Manage IT systems in multidivisional organizations

Large organizations are often organized as a set of smaller organizations (divisions) that have a certain amount of autonomy. In the area of Information Technology this can result in numerous problems; some examples are:

- Duplication: Several divisions may build systems that perform the same function.
- Incompatibility: It may be difficult for a system built within one division to exchange data with systems built by other divisions.
- Inconsistent user interfaces: Clients and suppliers may find it hard to deal with differences in the behaviour of IT systems in different divisions.
- Differing outsourcing policies: One division may perform a function "in-house" while others give contracts for those functions to external suppliers.
- Inconsistent standards: The IT field has many competing and overlapping standards. Each division might pick its own standards with the result that an organization will have conflicting policies.
- Security issues: One division may release information that another division treats as confidential or withhold information that another division should be able to access.

In short, Information systems engineers need to learn how to design and manage an "enterprise architecture" that allows the individual divisions and the complete organization to function efficiently.

References

Ardis, M., Budgen, D., Hislop, G.W., Offutt, J., Sebern, M., Visser, W., Nov. 2015. SE 2014: curriculum guidelines for undergraduate degree programs in software engineering. Computer 48 (11), 106–109. doi:10.1109/MC.2015.345.

Brooks, F.P., 1995. The Mythical Man-Month: Essays on Software Engineering, second ed. Addison Wesley, Reading, MA.

Buxton, J.N., Randell, B. (Eds.), 1969, Software Engineering Techniques: Report on a Conference sponsored by the NATO Science Committee, Rome, Italy, 27 to 31 October. Scientific Affairs Division, NATO, Brussels. April 1970, p. 164.

Computing Curricula, 2005. The Overview Report http://www.acm.org/education/education/curric_vols/CC2005-March06Final.pdf.

Glass, R.L., Ramesh, V., Vessey, I., June 2004. An analysis of research in computing disciplines. Comm. ACM 47 (6), 89–94. doi:10.1145/990680.990686.

Lethbridge, T.C., LeBlanc Jr., R.J., Kelley-Sobel, A.E., Hilburn, T.B., Díaz-Herrera, J.L., 2006. SE2004:recommendations for undergraduate software engineering curricula. IEEE Softw. 23 (6), 19–25. doi:10.1109/MS.2006.171, Nov.-Dec..

Lutz, M.J., Fernando, N., Vallino, J.R., August 2014. Undergraduate software engineering. Comm. ACM 57 (8), 52–58. doi:10.1145/2632361.

Naur, P., Randell, B. (Eds.), 1968, Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany, 7 to 11 October. Scientific Affairs Division, NATO, Brussels. January 1969, p. 231.

Parnas, D.L., 1998. Software engineering programmes are not computer science programmes. Ann. Softw. Eng. 6, 19–37 Reprinted (by request) in IEEE Software, Nov. - Dec. 1999, pp. 19-30.

The Joint Task Force for Computing Curricula, 2005. ACM Curricula Recommendations http://www.acm.org/education/curricula-recommendations.

[11] Lower quality data is data that is less reliable or less accurate than other data.

Dr David Lorge Parnas taught his first computer design course in 1960 and has been studying software design and development since 1969. He has won more than 25 awards for his contributions In 2007, Parnas was proud to share the IEEE Computer Society's one-time sixtieth anniversary award with the late computer pioneer Professor Maurice Wilkes of Cambridge University. Parnas received his B.S., M.S. and Ph.D. in Electrical Engineering from Carnegie Mellon University and honorary doctorates from the ETH in Zürich, the Catholic University of Louvain, the University of Italian Switzerland, and the Technische Universität Wien. He is licensed as a Professional Engineer in Ontario. Parnas is the author of more than 285 papers and reports. Many have been repeatedly republished and are considered classics. He was the founding director of the McMaster University Software Engineering programme (CEAB accredited). A collection of Dr. Parnas' early papers can be found in: Hoffman, D.M., Weiss, D.M. (eds.), "***Software Fundamentals: Collected Papers by David L. Parnas***", Addison-Wesley, 2001, 664 pgs., ISBN 0-201-70,369-6. Dr. Parnas is Professor Emeritus at McMaster University in Hamilton Canada, and the University of Limerick Ireland. He is President of Middle Road Software in Ottawa, Ontario.