SOFTWARE ENGINEERING IN CONTEXT

BY PANKAJ KAMTHAN

1. INTRODUCTION

Software engineering is not about right and wrong but only better and worse.
— Ellen Ullman

It's natural to try to **reduce something unfamiliar to something known**, especially if you're not motivated to investigate the unfamiliar. [...] It's good to be wary. But the **belittling dismissal** does not come from experience.
— John Vlissides

In this document, **pairwise comparison** between software engineering and **programming**, **computer science**, and **'conventional' engineering** is presented. A comparison can help **clarify the boundary** of the discipline of software engineering, and, in doing so, help improve the understanding of software engineering.

It is assumed[1] that these disciplines are pairwise **comparable**[2]. The comparison in each case is based on some criteria. The attention is less on the similarities and more on the differences.

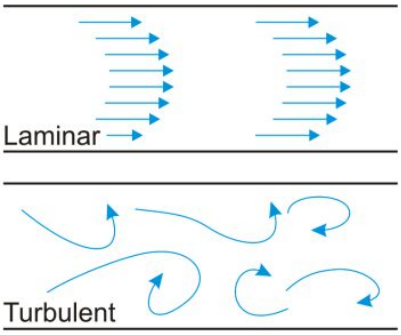
REMARKS

It could be noted that there are sub-areas of software engineering, such as ethnography, cost estimation, and project management that are **not** covered by any of the aforementioned disciplines.

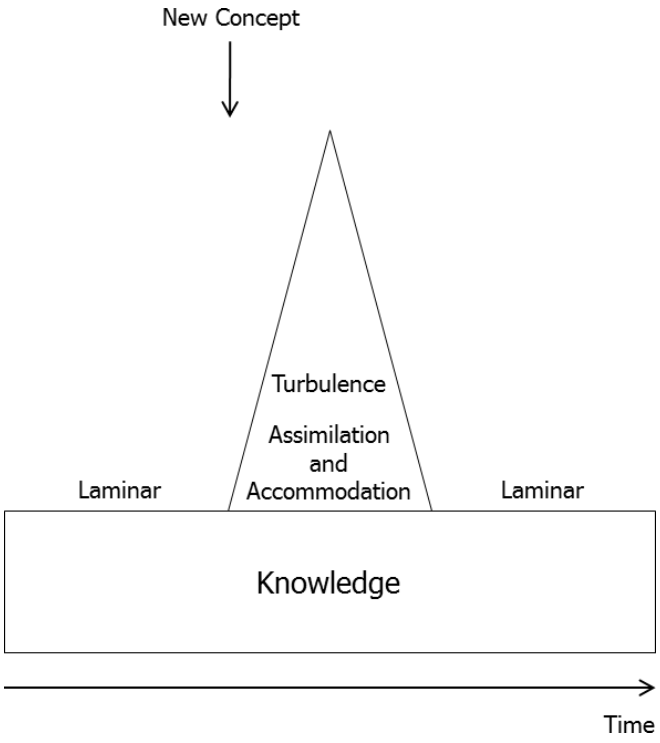
[1] This assumption can be justified based on empirical evidence.
[2] It is not automatic that any two given things are comparable. For example, in **APIs: A Strategy Guide**, it is asked: **"How Is an API Different from a Website?"** This is not a question to be asked since API and Website are not comparable.

2. THE INEVITABILITY OF COMPARISON

There are **theories of learning** [Hadjerrouit, 2005] which suggest that, upon initial exposure of a new concept, a comparison between that concept and other, more established, concepts is **inevitable**. This is illustrated in Figure 1.



(a)



(b)

Figure 1. (a) The difference between laminar and turbulent flows in a liquid. (Source: Wikipedia.) (b) The integration over time of a new concept in existing knowledge.

A comparison has a number of benefits: to further the understanding of software engineering as a discipline, to supply reasons for the difficulties in estimating the cost of software projects, and so on.

3. SOFTWARE AND SOFTWARE ENGINEERING

The term 'software' (like the term 'bit') was coined in the 1950s by John Tukey [Leonhardt, 2000]. In the ensuing decades, software has had profound impact on society. It has changed people's lives in many fundamental ways, **not always for the better** [Kitchin, Dodge, 2011; Spinellis, 2016; Cleland-Huang, 2016].

3.1. DEFINITION OF SOFTWARE AND SOFTWARE SYSTEM

There are a number of definitions of software. (The dictionary definitions of software are, in general, **not** useful.)

The next two definitions are from the 1980s and 1990s, respectively. They are similar, relatively specific, and dated:

Definition [Software] [IEEE, 1990; ISO/IEC/IEEE, 2010]. [A collection of] computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.

Definition [Software] [ITS[3]]. A set of computer programs, procedures, and associated documentation concerned with the operation of a data processing system.

The next definition is relatively general:

Definition [Software] [Wikipedia]. A collection of computer programs and related data that provide the instructions for telling a computer what to do and how to do it.

REMARKS

It follows from the previous definitions that "software" is a more encompassing concept than "computer program" and that software is about **multiplicity (plurality)**.

[3] URL: http://www.its.bldrdoc.gov/ .

3.2. DEFINITION OF SOFTWARE SYSTEM

Definition [Software System] [Sommerville, 2011]. A system consisting of "a number of separate programs, configuration files, which are used to set up these programs, system documentation, which describes the structure of the system, and user documentation, which explains how to use the system […]" [Sommerville, 2011, Section 1.1].

Therefore, "software system" is a more encompassing concept than "software".

REMARKS

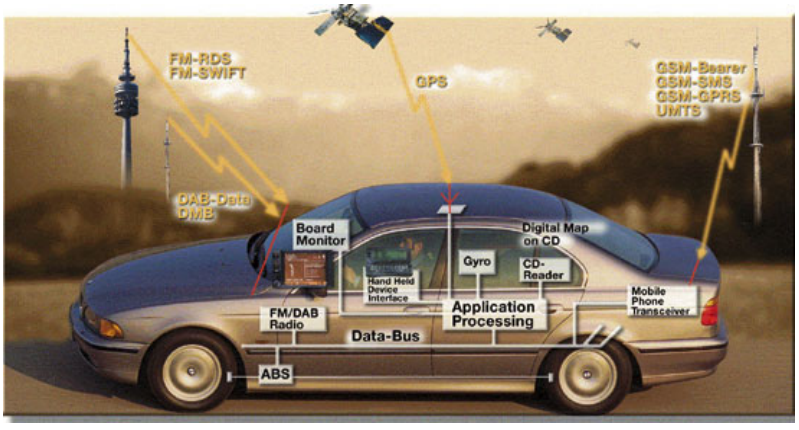
It could be noted that **"system software"** is "the computer software designed to provide services to other software" [Wikipedia].

3.3. DEFINITION OF SOFTWARE-INTENSIVE SYSTEM

The notion of a software-intensive system is a **generalization** of a software system, as suggested by the next definition:

Definition [Software-Intensive System] [Booch, 2006]. A system where software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole.

A software-intensive system involves some degree of **software-hardware interplay**, such as that found in large **distributed systems**, in small **embedded systems**, and **multicore systems** [Booch, 2006]. However, in a software-intensive system, software is the **major contributing factor** to cost, schedule, and risk [ISO/IEC/IEEE, 2010].



(Source: Google Images.)

In this document, the terms **'software'**, **'software system'**, and **'software-intensive system'** are **synonymous**, and therefore interchangeable, unless otherwise stated.

3.3. CLASSIFICATION OF SOFTWARE

There are **different types of software** and **different ways of classifying software** [Muffatto, 2006; Leveson, 2011; Sommerville, 2011, Section 1.1; Cesare, Xiang, 2012; Herraiz, Rodriguez, Robles, Gonzalez-Barahona, 2013].

3.4. DEFINITION OF SOFTWARE ENGINEERING

The term **'software engineering'** was coined in 1968 at a conference[4] sponsored by the **NATO Science Committee** [Bauer, 2010, Pages 118-119].

There are a number of definitions of software engineering, including the following that originates from a standard and is therefore authoritative:

Definition [Software Engineering] [IEEE, 1990; ISO/IEC/IEEE, 2010].
(1) The application of a **systematic**, **disciplined**, **quantifiable** approach to the **development**, **operation**, and **maintenance** of software; that is, the application of **engineering** to software.
(2) The study of approaches as in 1.

There are a number of interrelated 'elements' of software engineering, given by the **IEEE Software and Systems Engineering Standards Committee**, as illustrated in Figure 2.

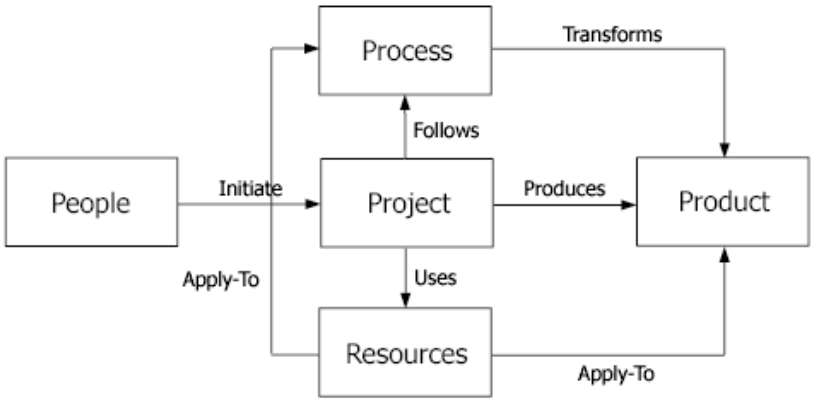


Figure 2. The basic elements of software engineering. (Source: IEEE.)

[4] URL: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/ .

4. SOFTWARE ENGINEERING AND OTHER DISCIPLINES

There are a number of disciplines that contribute, directly or indirectly, to software engineering [Kelly, 2015].

Table 1 provides a list of a number of disciplines on which software engineering relies upon, or overlaps with, for its existence and relevance. In certain cases, such as human-computer interaction and information technology, the relationship is **bidirectional**.

Software Engineering	• Biology • Business • Computer Engineering • **Computer Science** • **Engineering** • Ethnography • Economics • Human-Computer Interaction • Information Technology • Law • Linguistics • Management • **Mathematics** • Multimedia • Philosophy • Physics • **Programming** • Psychology • Sociology • Statistics

Table 1. A list of disciplines ancillary to software engineering.

Table 1 can be viewed in the other direction. There are signs that software engineering is **'returning the favor'** to fields it has **'borrowed from'** [Spinellis, 2015]: "software engineering has benefited mightily from research in fields ranging from electrical engineering and physics to mathematics and management science. […] Now, the time has come to [give] back to science and technology the knowledge software engineering has produced in the past half century".

PROBLEM (OR APPLICATION) DOMAIN

In addition to the disciplines listed in Table 1, a software system depends on the knowledge of its underlying (problem) **domain**.

For example, finance, in general, and **banking**, in particular, is the domain of a software system simulating an **Automated Teller Machine (ATM)**; accounting is the domain of an software for processing and filing **income tax**; **medicine** is the domain for **healthcare information system**; and so on.

In the last couple of decades or so, there has been an influx of several new domains such as **oil and gas**, **gambling**, and so on, and the list continues to grow.

In general, a software engineer is usually **not** trained in the knowledge of these domains.

REMARKS

It has been suggested that an **additional degree in the application domain** can help, and an example from the **medical domain** has been given in support of this argument [Laplante, 2007, Page 5].

However, for a number of reasons, the above suggestion is an **impractical solution**: obtaining another degree entails financial costs that can be prohibitive for some; obtaining another degree adds more time (possibly, years) that may not make a person any more marketable at the time of completion of the degree; and a person cannot be reasonably expected to pursue a degree in **every** application domain (that he or she may or may not be aware of a priori, but has to deal with during employment).

5. A COMPARISON OF PROGRAMMING AND SOFTWARE ENGINEERING

The similarities and differences between large-scale programming and software engineering have been addressed previously in an educational setting [Deimel, 1989; Marques, Ochoa, 2013].

The following **criteria** are used for comparing programming and software engineering:

- Artifacts
- Maintenance
- Management
- Problem
- Stakeholders

ARTIFACTS

Programming

In programming, a program can be **completed without any design, conceptual models, or documentation**. There may not be any intermediate artifacts. There may not be any deliverable except the program itself.

Software Engineering

In software engineering, a software system needs to be developed based on a **disciplined and systematic** approach. It requires the production of artifacts, including conceptual models and/or documentation, at each stage, regardless of the development methodology.

Indeed, the **significance of conceptual modeling** [Ambler, 2002] and **documentation** [Parnas, 2011] has been described elsewhere.

MAINTENANCE

> You learn to write as if to **someone else** because next year you will be "someone else."
> — Brian W. Kernighan and Philip J. Plauger

Programming

A program may not be written with maintenance as one of the **anticipated considerations**, and therefore, or even otherwise, may not be maintained. Indeed, the program may be **discarded** after use for a short period.

Software Engineering

A software system, almost invariably, has **life after development and delivery**.

For example, a distributed information system for a **bank** will **change** after delivery, as there are **changes in the software system's operating environment** (say, in response to relevant changes in the legal system; increase in the number of clients, and data related to those clients; changes over time in the expectations of clients; relevant changes in technology underlying the operating system; discovery of defects; and so on).

These changes need to be **anticipated and acknowledged**. In other words, it is almost always **necessary** that a software system be maintained [Herraiz, Rodriguez, Robles, Gonzalez-Barahona, 2013; Godfrey, Germán, 2014].

Indeed, maintenance needs to be as an **integral** part of development [Grubb, Takang, 2004]. In industrial software development, a **major portion (60-80%**, by some estimates) of the budget is dedicated to maintenance, as shown in Figure 3. It could be noted that quantification (that is, exact percentages or numbers themselves) are **less important** and qualification (that is, the message) is more important.

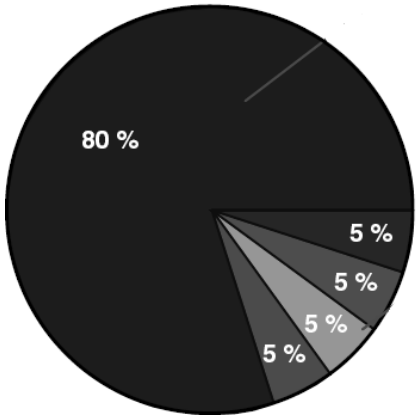


Figure 3. The cost of maintenance relative to other activities [Keyes, 2003].

REMARKS

In could be noted that programming and software engineering are **not mutually exclusive**.

Indeed, as shown in Figure 4, the **implementation** phase of a software process involves programming, as exemplified by the **Software Construction Knowledge Area** of the **Guide to the Software Engineering Body of Knowledge (SWEBOK)** [IEEE, 2014]. Indeed, by some estimates, programming constitutes **20-30%** of the overall software development effort [Kaniss, 2015].

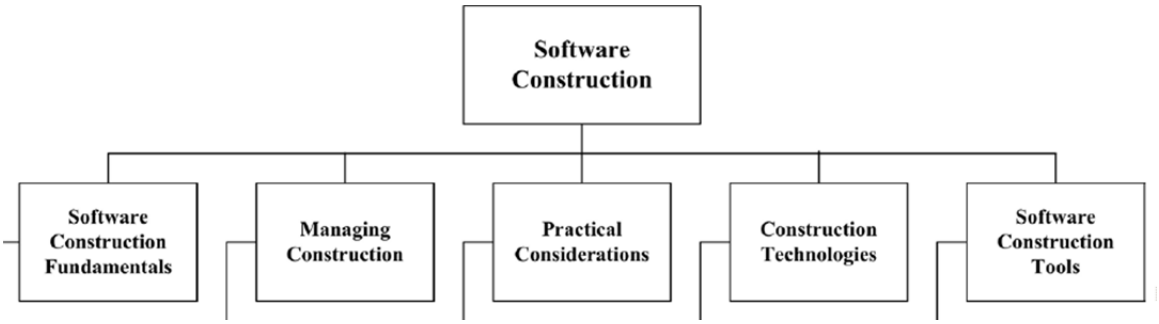


Figure 4. Software Construction is a Knowledge Area in the Guide to the Software Engineering Body of Knowledge (SWEBOK). (Source: SWEBOK [IEEE, 2014].)

MANAGEMENT

Programming

In programming, a program can be managed by a **single programmer**. A program can be completed in relatively **short time**, say, few hours or few days.

Software Engineering

In software engineering, a software system may need a **team** of several software engineers to be managed. In general, **prima donnas** are **discouraged**. (A team has a significant social aspect that needs to be managed. A team requires **communication and collaboration** that is usually realized by regular **meetings**.)

The development of a software system can take relatively long time, say, few months or, in some cases, few years, to be completed.

PROBLEM

Computers are useless. They can only give you answers.
— Pablo Picasso

Programming

A program is a solution to a problem.

In programming, the problem may be **given**. For example, in a basic programming course, the problem description is usually given.

Software Engineering

A software system is a solution to a problem. Indeed, according to one of the theories, software engineering can be viewed as a **problem-solving discipline** [Tekinerdogan, Aksit, 2011; Hall, Rapanotti, 2013; Hall, Rapanotti, 2015]. However, the difference between the problem (and solution) corresponding to a program and a software system is that of **scale**. Usually, a software system aims to solve a **relatively larger problem**.

As an analogy, if programming addresses the problem of building a **dog house**, then software engineering addresses the problem of building a **family house** or, in some cases, even a **skyscraper** [Kaniss, 2015].

In software engineering, the problem is **almost never known** a priori, at least not in its entirety. In certain cases, especially for interactive software systems, the problem is **unclear** at the **inception** of the software project, as illustrated in Figure 5. Therefore, an important aspect of software development is the **elicitation** of the problem (that is, **requirements**).

Figure 5. The nature of requirement engineering processes for typical interactive software systems.

In fact, in software engineering, the problem is a **"wicked problem"** [DeGrace, Stahl, 1990; Carlshamre, 2002; Nerur, Balijepally, 2007; Sommerville, 2011, Section 10.1.3; Janes, Succi, 2014, Section 1.2].

A **wicked problem** is a problem that is "difficult or impossible to solve because of incomplete, contradictory, and changing requirements that are often difficult to recognize" [Wikipedia]. (In contrast to a wicked problem is a **tame problem**, such as those that occur in **mathematics, chess, or puzzle solving**.) In particular, a wicked problem is not expressible entirely in any single linguistic system and does not have a mathematical proof [Hall, Rapanotti, 2013].

In software engineering, solutions of wicked problems must involve "discussion, consensus, iterations, and acceptance of change as a normal part of the process".

STAKEHOLDERS

There are a number of definitions of stakeholder, including the following:

Definition [Stakeholder] [Wiegers, 2006, Chapter 2]. [An] individual or group who is actively involved in the project, who is affected by the project, or who can influence its outcome.

Programming

In programming, a program could be aimed for **personal use** or for a **known client or user**.

For example, a program could be a utility for organizing personal files, or it could be part of an assignment by a student for a teacher.

It is **clear** what is to be developed. (If not, clarity can be sought relatively easily.)

The responsibilities of a programmer, in principle, **end** when the compiler says **"No Errors"**.

Software Engineering

In software engineering, a software system is usually aimed for specific clients or users that are **not necessarily known** a priori.

For example, a software system could be a shopping system for a client in a geographical location different from that of development, and its users could be **globally distributed** [Sangwan, Bass, Mullick, Paulish, Kazmeier, 2007].

It is crucial that exact **needs** of a client be **elicited before development** and **verified after development** [ISO/IEC/IEEE, 2011].

The responsibilities of a software engineer, in principle, **start** when the compiler says **"No Errors"**.

Figure 6 summarizes part of the previous argument.

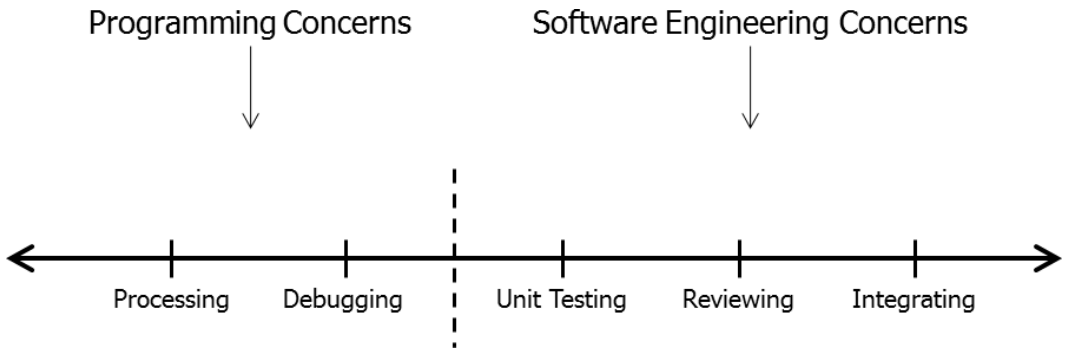


Figure 6. The expectations in software engineering go beyond that in programming.

REMARKS

- In software engineering, the considerations of the **'others'** are deemed more important than the considerations of an individual. For example, this is important to reach a consensus on the terminology of a domain. If the individual is a team member, these 'others' include the rest of the **team members**, **client** (if any), and **users**.

  In other words, **altruism and empathy** are necessary tenets of a software engineer. The emphasis on the collective, rather than on an individual, is especially prominent in **agile methodologies**.

For example, **Collective Ownership** is one of the practices of **Extreme Programming (XP)** as described in [Beck, 2000], and **Shared Code** is one of the corollary practices of XP as described in [Beck, Andres, 2005]. These practices do **not** occur automatically, and require certain **temperament and skills** for successful execution.

It has been predicted [Boehm, 2011] that **attention towards users** and their needs will continue to be important determinant, and will shape all aspects of software development, including business, organizational, and technical aspects.

- It could be said that in software engineering all **major artifacts are produced for the consumption by 'others'**, as shown in Figure 7. For example, a list of possible artifacts that apply to most software development methodologies and possible stakeholders is given in Table 2.

Software Artifact	Stakeholder(s)
Domain Model	Client, Requirements Engineer
Software Requirement	Client, Project Manager, Designer, Programmer, Tester
Source Code	Maintainer, Quality Assurance Engineer, Reviewer, Tester

Table 2. A collection of software artifacts and their stakeholders.

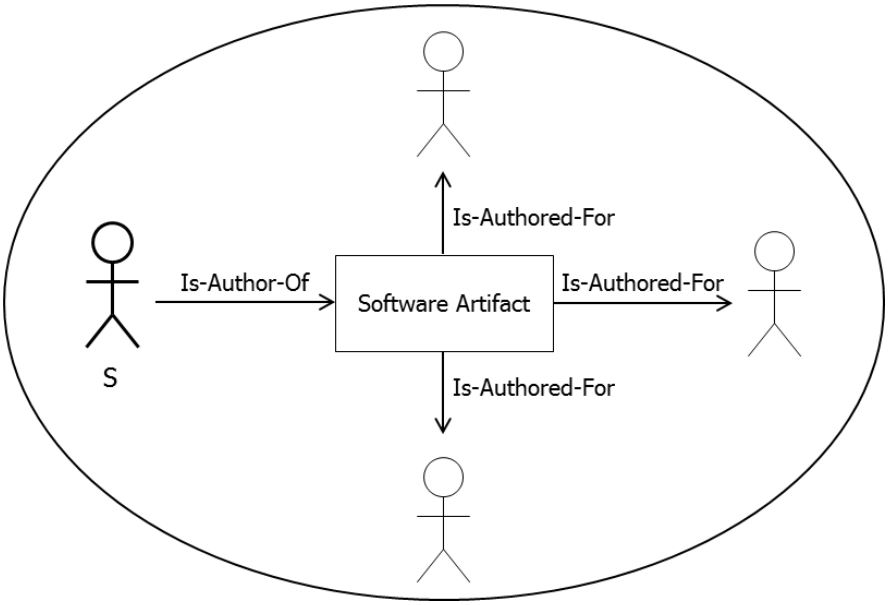


Figure 7. A software artifact is **authored for others**, by being **mindful of others**.

6. A COMPARISON OF COMPUTER SCIENCE AND SOFTWARE ENGINEERING

A scientist builds in order to learn; an engineer learns in order to build.
— Frederick P. Brooks, Jr.

The similarities and differences between computer science and software engineering have been addressed [Parnas, 1998; Engle Jr., 1989; Tockey, 2004; Miller, Voas, 2008; Connell, 2009; Frezza, 2010; Briand, 2012; Hazzan, Dubinsky, 2014].

It has been pointed out that "science is the pursuit of knowledge, and engineering is the application of that knowledge for the benefit of people" [Tockey, 2004].

Figure 8 presents the relationship between concepts in **biology and software engineering**, in general, and data structures and (C++) programming, in particular.

1.	Introduction to Cells	1.	Software Engineering Principles and C++ Classes
2.	Chemical Components of Cells	2.	Object-Oriented Design (OOD) and C++
3.	Energy, Catalysis, and Biosynthesis	3.	Pointers and Array-Based Lists
4.	Protein Structure and Function	4.	Standard Template Library (STL) I
5.	DNA and Chromosomes	5.	Linked Lists
6.	DNA Replication, Repair, and Recombination	6.	Recursion
7.	From DNA to Protein: How Cells Read the Genome	7.	Stacks
8.	Control of Gene Expression	8.	Queues
9.	How Genes and Genomes Evolve	9.	Search Algorithms
10.	Manipulating Genes and Cells	10.	Sorting Algorithms
11.	Membrane Structure	11.	Binary Trees
12.	Membrane Transport	12.	Graph Algorithms
13.	How Cells Obtain Energy from Food	13.	Standard Template Library (STL) II
14.	Energy Generation in Mitochondria and Chloroplasts	A.	Reserved Words
15.	Intracellular Compartments and Transport	B.	Operator Precedence
16.	Cell Communication	C.	Character Sets
17.	Cytoskeleton	D.	Operator Overloading
18.	The Cell Division Cycle	E.	Header Files
19.	Genetics, Meiosis, and the Molecular Basis of Heredity	F.	Additional C++ Topics (Inheritance, Pointers, and Virtual Functions)
20.	Tissues and Cancer	G.	Problem Solving Using Object Oriented Methodology
—		H.	C++ for Java Programmers
—		I.	References for Further Study
—		J.	Answers to Odd-Numbered Exercises

Figure 8. There are similarities between science (biology) and (software) engineering, as indicated by the topical comparison of the Table of Contents of "Essential Cell Biology" (left) and "Data Structures Using C++" (right). (Source: [Frezza, 2010].)

The following **criteria** are used for comparing computer science and software engineering:

- Focus
- Human Component
- Plurality

FOCUS

Computer Science

In computer science, the focus is on the **theoretical** aspects of software development. These include introduction of necessary discrete mathematical structures, automata theory, and the development of data structures, algorithms, and programming languages.

Software Engineering

In software engineering, the focus is on the **practical** aspects of software development. These include the development of software management processes, software quality models, and software metrics.

The practical aspects rest on theoretical aspects. Indeed, software engineering should be "treated as a **scientific discipline**" [Wohlin, Runeson, Höst, Ohlsson, Regnell, Wesslén, 2012, Page 6].

However, it should be accepted and understood that software engineering is **not linear** (same inputs can lead to different outputs) and **not purely science** (software process involves continual choices among multiple viable alternatives) [Marques, Ochoa, 2013].

HUMAN COMPONENT

Computer Science

A topic, especially if it is related to mathematics, may **not** have a human component. For example, such topic could be **automata theory, discrete mathematics, or numerical analysis**.

Software Engineering

A number of topics, such **project management** and **usability engineering**, have an **inherent human component** [Connell, 2009; Marques, Ochoa, 2013].

REMARKS

It is evident that computer science is **necessary**, but **not sufficient**, to understand and overcome the problems that are faced in software engineering [Herbsleb, 2005; Connell, 2009; Rost, Glass, 2011].

PLURALITY

Computer Science

Usually, there is **one and only one correct solution** to a given problem.

Software Engineering

> In design [...] most of the time and effort is spent in generating the alternatives, which aren't given at the outset. [...] The idea that we start out with all the alternatives and then choose among them is wholly unrealistic.
> — Herbert A. Simon

Usually, there can be **many** correct solutions (say, designs) to a given problem.

In fact, **collectively exploring and seeking diverse solutions** is inherent. (This calls upon an approach that encourages **divergent thinking and collaborative learning** in software engineering education [Offutt, 2013].)

7. A COMPARISON OF 'CONVENTIONAL' ENGINEERING AND SOFTWARE ENGINEERING

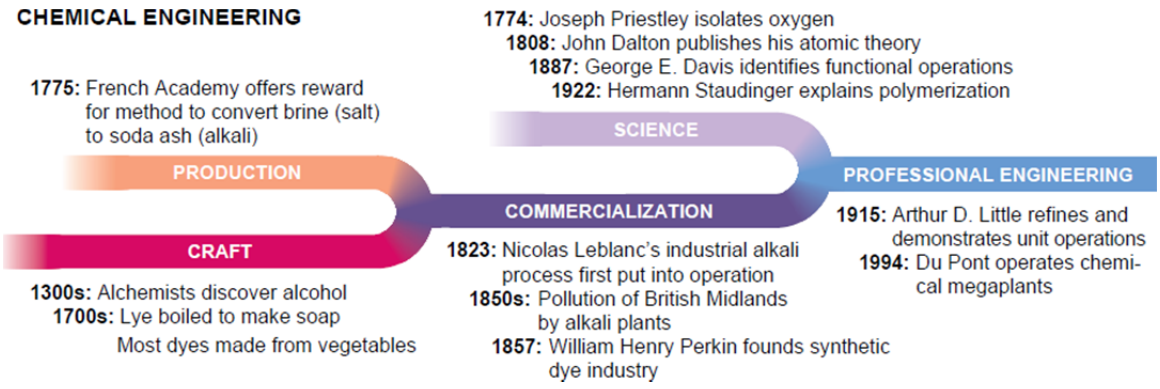
If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.
— Gerald Weinberg

Software is not limited by physics, like buildings are. It is limited by imagination, by design, by organization. In short, it is limited by properties of people, not by properties of the world.
— Martin Fowler

The term 'conventional' engineering is used in the sense of non-software engineering, often related to **manufacturing**. These include **chemical engineering, civil engineering, electrical engineering, industrial engineering, mechanical engineering**, and their cognate disciplines.

The relationship between 'conventional' engineering and software engineering has been examined [Agresti, 1981; Lea, 1994; Holloway, 1999; Kruchten, 2004; McConnell, 2004, Chapter 4; Cockburn, 2007; Denning, Riehle, 2009; Jackson, 2011; Kendrick, 2011, Problem 13; Briand, 2012; Offutt, 2013].

The engineering disciplines share **common stages** in their evolution. Figure 9 shows **parallels** between the **evolution of chemical engineering and software engineering** [Gibbs, 1994].



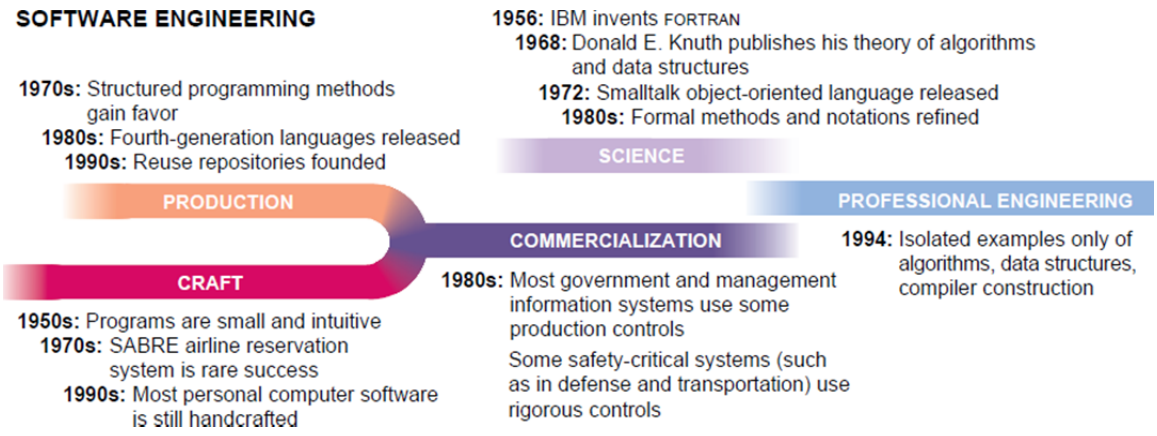


Figure 9. There are similarities between the evolution of chemical engineering and software engineering. (Source: [Gibbs, 1994].)

The following **criteria** are used for comparing 'conventional' engineering and software engineering:

- Conformance
- Delivery
- Laws of Nature
- Mathematics
- Maturity
- Production
- Spatial Relationships
- Maintenance

CONFORMANCE

Conventional Engineering

In 'conventional' engineering disciplines, the notion of conformance is (usually) **interval-based** [Binder, 1997].

For example, let the radius of **automobile tire** be set to 30 cm. Then, tires of both 29.0001 cm and 30.0001 cm radii may be considered acceptable. This is by **necessity rather than choice** since it not always physically possible to attain an exact measurement. Moreover, the error (absolute or relative) is **negligible**.

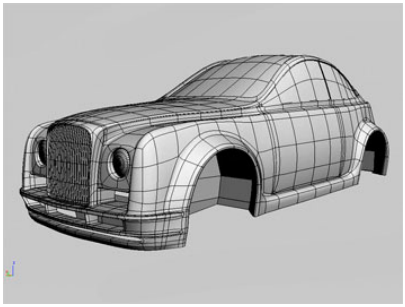
Software Engineering

In software engineering, the notion of conformance is (usually) **point-based**.

DELIVERY

Conventional Engineering

It is **uncommon** to deliver a product that is **'incomplete'** in some fundamental way, as such an 'incomplete' product will not be usable. (There are exceptions.)



(Source: Google Images.)

It is **necessary** to deliver the **source** of the product.

It is not common to deliver the products of 'conventional' engineering disciplines with known 'defects'; it is even less common to announce them.

Software Engineering

It is **not uncommon** to deliver a product that is 'incomplete' in some fundamental way, and such an 'incomplete' product can be usable.

For example, an electronic shopping system, S, may only sell books and only allow credit card payment initially. S may evolve to sell other items and allow other means of payment, both without a consumer's knowledge. The **amazon.com** is a prime example.

For another example, consider a diagramming program, such as **Microsoft Visio**. Microsoft Visio is considered usable, even though the support for the Unified Modeling Language (UML), Version 2.5, in Microsoft Visio is **fundamentally inadequate**.

For a contrasting example, a software system S may have commercial (S_C) and non-commercial (S_{NC}) incarnations, such that the functionality of **S_{NC} is a proper subset of that of S_C**. (To be specific, S_{NC} could be a 'trial' version made available so that the users can try it out, perhaps even provide feedback based on their experience with it, in the hope that some, if not all, of those users will purchase S_C.)

It is **not necessary to deliver the source** of the product. For example, one can deliver only the binary that runs on the operating system of user's device.

[The] average customer of the computing industry has been served so poorly that he expects his system to crash all the time, and we witness a massive worldwide distribution of bug-ridden software for which we should be deeply ashamed.
— Edsger W. Dijkstra

If people really knew how software got written, I'm not sure they'd give their money to a bank or get on an airplane ever again.
— Ellen Ullmann

In software engineering, systems are (sometimes, proudly) delivered, knowing that they have defects[5]. It is common by the owner, including by a large company in Seattle, to publicly announce the list of defects and, in some cases, **make available patches** to fix those defects.

Indeed, patches for software are routine[6] rather than rare.

LAWS OF NATURE

[5] To check the validity of the assertion, a query string such as "known [name of software] bugs" could be used on a global search engine. A collection of positive response(s) is more than likely.
[6] URL: http://www.softwarepatch.com/ .

Software is **not natural**. It does not age, rust, decay, break, melt, evaporate, vibrate, or float. The laws of nature do not apply to software.
— Richard Bechtold

Conventional Engineering

The products of 'conventional' engineering disciplines are governed by **physical laws** (or **laws of nature**[7]), and therefore are **impacted** by these laws.

For example, a bridge can be **permanently damaged** by a hurricane; a house **decays** over time, even if uninhabited; an **automobile tire loses air** over time, even if untouched by humans; the molecular structure of a beer bottle, if broken (usually, by humans), is **altered irreversibly**; and so on.

Software Engineering

The discipline of software engineering is **impervious** to laws of nature: conventional physical laws do not apply to software systems.

For example, environmental factors such as **temperature, pressure, and humidity**, are not relevant to software systems.

MATHEMATICS

Conventional Engineering

It is common for students of 'conventional' engineering to study a specific collection of basic topics in **discrete and continuous mathematics**.

For example, such topics include functions, Boolean logic, complex analysis, Fourier and Laplace transforms, matrices, numerical analysis, and differential equations, to name a few. These topics are aggregated and collectively labeled as **'engineering mathematics'**.

Software Engineering

It is not what was taught in the mathematics class that was important; it's the fact that it was mathematical.
— Keith Devlin

[7] These include, but are not limited to, law of gravitation, laws of motion, ideal gas laws, and so on.

$$\{P\}\ C\ \{Q\}$$

There are certain basic topics in discrete mathematics that are **directly or indirectly relevant** to software engineering.

For example, basic topics in discrete mathematics include **logic, sets, functions, relations, combinatorics, matrices, and graphs**, to name a few. There is **overlap** among the topics in discrete mathematics relevant to 'conventional' engineering and software engineering.

- **Direct Relevance of Mathematics in Software Engineering.** The direct relevancy is primarily due to the origins of software engineering in computer science and computer programming.

- **Indirect Relevance of Mathematics in Software Engineering.** The indirect relevancy is from (1) the formation of ability to infer and reason logically, and (2) the formation of basis to learn advanced mathematics if necessary. For example, understanding of **mathematical induction** is necessary for proving the **correctness of certain iterative and recursive algorithms**, understanding of **predicate logic** is necessary for **certain formal specification languages and for description logics**, and so on.

In comparison, the use of **advanced continuous mathematics** is not common in software engineering. However, it may be required by certain problem domains. For example, such is the case when producing **mathematical software** for **numerically or symbolically solving problems** in economics, science, and engineering.

REMARKS

It is important for engineers of all types to be aware of the **limits of mathematics for digital computation**. In particular, many established results in mathematics (such as those related to arithmetic) do **not** hold on digital computers [Gander, Gander, Kwok, 2014].

MATURITY

Conventional Engineering

The 'conventional' engineering disciplines have a long history, of the order of several decades, even centuries. They are considered established.

Software Engineering

In comparison, software engineering is a (relatively) new discipline and has a short history, of the order of a few decades [Campbell-Kelly, 2003; Mahoney, 2004; Bauer, 2010; Rajlich, 2012; O'Regan, 2013; **Díaz-Herrera, Freeman, 2014**; Jones, 2014; Tedre, 2015]. It is therefore unsurprising that **controversies, debates, and myths** linger [Berry, Lawrence, 1998; Bryant, 2000; McBreen, 2002; Glass, 2003].

For example, it is pointed out that, as a **problem-solving discipline**, software engineering has yet to evolve [Tekinerdogan, Aksit, 2011].

PRODUCTION

Conventional Engineering

In 'conventional' engineering, a product is **built**. In 'conventional' engineering disciplines, especially those related to manufacturing, there is **repeated construction** of (seemingly) **identical** products.

It can be expected that a 'conventional' engineering process has the following phases: **requirements, design, implementation, and testing**.

It is interesting to note that the **reliance on software in 'conventional' engineering** has increased over the years. (Therefore, there is a **feedback loop** between 'conventional' engineering and software engineering.) For example, it has been pointed out that "60-90% of advances in the automotive domain nowadays are due to software systems" [Trendowicz, Jeffery, 2014, Page 3].

Software Engineering

In software engineering, a product is (usually) **grown** [Monson-Haefel, 2009] through a **creative** process [Tekinerdogan, Aksit, 2011]. It is possible to follow the steps of a software process, but **not repeat it mechanically**.

In software engineering, there is always a **canonical (unique) product**, copies of which can be made for distribution. There is no (technical) limit to the number of copies of software at any given point in time, and so the notion of **'inventory'** does not apply. (The notion of inventory does, however, apply to packaged software, such as **commercial-off-the-shelf (COTS)** software, sold in computer stores.)

The product has a number of **unique defining characteristics**:

- **Software is Invisible.** This makes it challenging to locate "defects" and to determine the extent of the implications of those "defects". It also poses a challenge to risk management.

- **Software is Intangible.** This means "software does not wear out but the situation often changes in a way that makes the software no longer completely fit for its original intended function" [Gotterbarn, 2010]. The problems underlying software project may become apparent only after a considerable time has lapsed [Janes, Succi, 2014, Section 1.3].

- **Software is Malleable.** This means software can take different forms, both good and bad [Moor, 1985].

- **Software is Changeable.** This means software can be easily changed by almost anybody, for better or worse [Bain, 2008, Chapter 3]. In fact, as soon as 'new' software is executed, it can change, say, by logging usage data.

- **Software is Discontinuous.** This means a small change in the input to software can lead to an unexpected (and, sometimes, incomprehensible) large changes in the output, not necessarily for the better. For example, consider a password entry that is off by a single character, a missing closing curly bracket (}) in the use of T_EX, and so on.

- **Software is Weightless.** This (software not weighing anything) can make implementers 'callous' about the use of space. (In contrast, there is considerable attention to the kind of material used in certain engineering disciplines, such as aeronautical engineering, automobile engineering, and civil engineering. For example, the power-to-weight ratio, and therefore the handling, speed, and acceleration, of a car depend on the material being an iron alloy or carbotanium.)

- **Software is in the 'Holes'.** This is assuming that software is being written and archived on **punched cards**, and in its early days it was.

The following anecdote summarizes these characteristics.

Barry Boehm's ACM Fellow Profile[8] [Hey, Pápay, 2015]:

> This is a story that I heard second-hand at TRW, but it may be an urban myth. It is about a weights engineer on a spacecraft. He accounts for all of the weight on a spacecraft.
>
> The weights engineer came to a software engineer and said, "according to budget, there is $3,000,000 accounted for the software and I want to know how much it weighs." The software engineer replied, "Nothing." The weights engineer said he wanted that kind of a job. "You get paid well to produce something that weighs nothing."
>
> A week later, the weights engineer came back with a deck of cards. "Is there any software in here?" he asked. The software engineer said, "Yes." **The weights engineer said he would weigh the deck to determine the weight of the software. The software engineer said, "You have it all wrong; we only use the holes."**

It can be expected that a typical software engineering process has at least the following phases: **requirements, design, implementation, and testing**.

There are a number of movements in software development processes that are based on the (positive) lessons learned from **automobile manufacturing** processes.

For example, **Lean Software Development** [Poppendieck, Poppendieck, 2007; Rich, Bateman, Esain, Massey, Samuel, 2007], which overlaps with **agile software development and sustainable software development**, is based on the following **principles**:

1. Eliminate Waste
2. Amplify Learning
3. Decide As Late As Possible
4. Deliver As Fast As Possible
5. Empower The Team
6. Build Integrity In
7. See The Whole

[8] URL: http://www.sigsoft.org/SEN/boehm.html .

SPATIAL RELATIONSHIPS

Conventional Engineering

In ‘conventional’ engineering disciplines, there is a possibility of non-proximal distribution of parts during development. For example, this is the case in **sub-contracting** or outsourcing.

However, a **non-proximal distribution of parts** after delivery is **not** possible. This is because the parts of the final product are closely related. For example, this is the case for a building or a car.

Software Engineering

In software engineering, there is a possibility of non-proximal distribution of files constituting an artifact, both during development and after delivery. For example, this is the case in **global software development** during development and for **distributed software systems** after delivery.

MAINTENANCE

Conventional Engineering

In ‘conventional’ engineering disciplines, such as manufacturing, there is **cost associated with shipping upgrades**. In some cases, there may not be any choice but to **ship the product** to the manufacturer.

Software Engineering

In software engineering, there is **no cost of shipping upgrades**, at least not anymore. (The use of the Internet and Web Services has changed this dramatically.) In general, the **product is not shipped** to the manufacturer.

It has been pointed out that **“skyscrapers are not scalable”** [Monson-Haefel, 2009]. For example, once constructed, a skyscraper is not supposed to change its height or location. Therefore, **not** all civil engineering analogies apply to software engineering.

8. INFORMATION TECHNOLOGY AND SOFTWARE ENGINEERING

There are a number of definitions of information technology (IT), including the following:

Definition [Information Technology] [WordNet]. The branch of engineering that deals with the use of computers and telecommunications to retrieve and store and transmit information.

In [Miller, Voas, 2008], it has been suggested that a software engineer is a kind of IT professional. This would be the case if a software engineer would be dealing **exclusively** with IT.

However, in general, that is **not** the case. For example, **Learn to Estimate** is one of the **97 Things Every Programmer Should Know** [Henney, 2010], and therefore estimation is relevant to a software engineer, but estimation is **not** (about technology, and therefore is not) subsumed by IT.

9. CONCLUSION

Our civilization runs on software.
— Bjarne Stroustrup

To pursue and be successful in software engineering requires a **different way of thinking**.

There are a number of **competing interests** in software development, including **human, social, political, commercial, organizational, managerial, and technical interests** that need to be **balanced**.

In the practice of software engineering, it is striking this **balance** that continues to be a challenge, and is likely to remains so in the foreseeable future.

ACKNOWLEDGEMENT

The author is grateful to Peter Grogono for inspiration, sharing knowledge, and useful discussions. The inclusion of images from external sources is only for non-commercial educational purposes, and their use is hereby acknowledged.

REFERENCES

[Agresti, 1981] Software Engineering as Industrial Engineering. By W. W. Agresti. ACM SIGSOFT Software Engineering Notes. Volume 6. Number 5. 1981. Pages 11-13.

[Ambler, 2002] Agile Modeling. By S. W. Ambler. John Wiley and Sons. 2002.

[Bain, 2008] Emergent Design: The Evolutionary Nature of Professional Software Development. By S. L. Bain. Addison-Wesley. 2008.

[Bauer, 2010] Origins and Foundations of Computing. By F. L. Bauer. Springer-Verlag. 2010.

[Beck, 2000] Extreme Programming Explained: Embrace Change. By K. Beck. Addison-Wesley. 2000.

[Beck, Andres, 2005] Extreme Programming Explained: Embrace Change. By K. Beck, C. Andres. Second Edition. Addison-Wesley. 2005.

[Berry, Lawrence, 1998] Requirements Engineering. By D. M. Berry, B. Lawrence. IEEE Software. Volume 15. Issue 2. 1998. Pages 26-29.

[Binder, 1997] Can a Manufacturing Quality Model Work for Software? By R. V. Binder. IEEE Software. Volume 14. Issue 5. 1997. Pages 101-102, 105.

[Boehm, 2011] Some Future Software Engineering Opportunities and Challenges. By B. Boehm. In: The Future of Software Engineering. S. Nanz (Editor). Springer-Verlag. 2011. Pages 1-32.

[Booch, 2006] The Accidental Architecture. By G. Booch. IEEE Software. Volume 23. Issue 3. 2006. Pages 9-11.

[Briand, 2012] Embracing the Engineering Side of Software Engineering. By L. Briand. IEEE Software. Volume 29. Number 4. 2012. Pages 93-96.

[Bryant, 2000] Metaphor, Myth and Mimicry: The Bases of Software Engineering. By A. Bryant. Annals of Software Engineering. Volume 10. Numbers 1-4. November 2000. Pages 273-292.

[Campbell-Kelly, 2003] From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry. By M. Campbell-Kelly. The MIT Press. 2003.

[Carlshamre, 2002] Release Planning in Market-Driven Software Product Development: Provoking an Understanding. By P. Carlshamre. Requirements Engineering. Volume 7. Number 3. 2002. Pages 139-151.

[Cesare, Xiang, 2012] Software Similarity and Classification. By S. Cesare, Y. Xiang. Springer Science+Business Media. 2012.

[Cleland-Huang, 2016] Requirements That Reflect Social Responsibility. By J. Cleland-Huang. IEEE Software. Volume 33. Number 1. 2016. Pages 109-111.

[Cockburn, 2007] What Engineering has in Common With Manufacturing and Why It Matters. By A. Cockburn. CrossTalk. April 2007. Pages 4-7.

[Connell, 2009] Software Engineering ≠ Computer Science. By C. Connell. Dr. Dobb's Journal. June 2009.

[DeGrace, Stahl, 1990] Wicked Problems, Righteous Solutions: A Catalog of Modern Engineering Paradigms. By P. DeGrace, L. H. Stahl. Yourdon Press. 1990.

[Deimel, 1989] Programming and Its Relation to Computer Science Education and Software Engineering Education. By L. E. Deimel. The 1989 SEI Conference on Software Engineering Education. Pittsburgh, U.S.A. July 18-21, 1989.

[Denning, Riehle, 2009] The Profession of IT: Is Software Engineering Engineering? By P. J. Denning, R. D. Riehle. Communications of the ACM. Volume 52. Issue 3. 2009. Pages 24-26.

[Díaz-Herrera, Freeman, 2014] Discipline of Software Engineering: An Overview. By J. L. Díaz-Herrera, P. A. Freeman. In: Computing Handbook, Volume I: Computer Science and Software Engineering, Third Edition. By T. Gonzalez, J. Díaz-Herrera, A. Tucker (Editors). CRC Press. 2014. Pages 1909-1928.

[Engle Jr., 1989] Software Engineering is Not Computer Science. By C. B. Engle Jr. The 1989 SEI Conference on Software Engineering Education. Pittsburgh, U.S.A. July 18-21, 1989.

[Frezza, 2010] Computer Science: Is It Really the Scientific Foundation for Software Engineering? By S. T. Frezza. Computer. Volume 43. Number 8. 2010. Pages 98-101.

[Gander, Gander, Kwok, 2014] Scientific Computing: An Introduction using Maple and MATLAB. By W. Gander, M. J. Gander, F. Kwok. Springer International Publishing. 2014.

[Gibbs, 1994] Software Chronic Crisis. By W. W. Gibbs. Scientific American. Volume 271. Number 3. 1994. Pages 86-95.

[Godfrey, Germán, 2014] On the Evolution of Lehman's Laws. By M. W. Godfrey, D. M. Germán. Journal of Software: Evolution and Process. Volume 26. Number 7. 2014. Pages 613-619.

[Glass, 2003] Facts and Fallacies of Software Engineering. By R. L. Glass. Addison-Wesley. 2003.

[Gotterbarn, 2010] Perfection is Not 'Good Enough': Beyond Software Development. By D. Gotterbarn. ACM Inroads. Volume 1. Number 4. 2010. Pages 8-9.

[Grubb, Takang, 2004] Software Maintenance: Concepts and Practice. By P. Grubb, A. A. Takang. World Scientific. 2004.

[Hadjerrouit, 2005] Constructivism as Guiding Philosophy for Software Engineering Education. By S. Hadjerrouit. ACM SIGCSE Bulletin. Volume 37. Issue 4. 2005. Pages 45-49.

[Hall, Rapanotti, 2013] Beauty in Software Engineering. By J. G. Hall, L. Rapanotti. Computer. Volume 46. Number 2. 2013. Pages 85-87.

[Hall, Rapanotti, 2015] A General Theory of Engineering: Thinking Bigger than Software. By J. G. Hall, L. Rapanotti. Technical Report No. 2015/01. Department of Computing. The Open University. January 28, 2015.

[Hazzan, Dubinsky, 2014] Agile Anywhere: Essays on Agile Projects and Beyond. By O. Hazzan, Y. Dubinsky. Springer. 2014.

[Henney, 2010] 97 Things Every Programmer Should Know: Collective Wisdom from the Experts. By K. Henney (Editor). O'Reilly Media. 2010.

[Herbsleb, 2005] Beyond Computer Science. By J. D. Herbsleb. The Twenty Seventh International Conference on Software Engineering (ICSE 2005). St. Louis, U.S.A. May 15-21, 2005.

[Herraiz, Rodriguez, Robles, Gonzalez-Barahona, 2013] The Evolution of the Laws of Software Evolution: A Discussion Based on a Systematic Literature Review. By I. Herraiz, D. Rodriguez, G. Robles, J. M. Gonzalez-Barahona. ACM Computing Surveys. Volume 46. Number 2. 2013. Pages 28:1-28:28.

[Hey, Pápay, 2015] The Computing Universe: A Journey through a Revolution. By T. Hey, G. Pápay. Cambridge University Press. 2015.

[Holloway, 1999] From Bridges and Rockets, Lessons for Software Systems. By C. M. Holloway. NASA Langley Research Center. Hampton, U.S.A. 1999.

[IEEE, 1990] IEEE Standard 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology. The Institute of Electrical and Electronics Engineers (IEEE) Computer Society. 1990.

[IEEE, 2014] Guide to the Software Engineering Body of Knowledge (SWEBOK) Version 3.0. The Institute of Electrical and Electronics Engineers (IEEE) Computer Society. 2014.

[ISO/IEC/IEEE, 2010] ISO/IEC/IEEE 24765:2010. Systems and Software Engineering -- Vocabulary. The International Organization for Standardization (ISO)/The International Electrotechnical Commission (IEC)/IEEE Computer Society. 2010.

[ISO/IEC/IEEE, 2011] ISO/IEC/IEEE 29148:2011. Systems and Software Engineering -- Life Cycle Processes -- Requirements Engineering. The International Organization for Standardization (ISO)/The International Electrotechnical Commission (IEC)/The Institute of Electrical and Electronics Engineers (IEEE) Computer Society. 2011.

[Jackson, 2011] Engineering and Software Engineering. By M. Jackson. In: The Future of Software Engineering. S. Nanz (Editor). Springer-Verlag. 2011. Pages 100-114.

[Janes, Succi, 2014] Lean Software Development in Action. By A. Janes, G. Succi. Springer-Verlag. 2014.

[Jones, 2014] The Technical and Social History of Software Engineering. By C. Jones. Addison-Wesley. 2014.

[Kaniss, 2015] What is a Software Engineer? By A. Kaniss. CrossTalk. May/June 2015. Pages 20-21.

[Kelly, 2015] Scientific Software Development Viewed as Knowledge Acquisition: Towards Understanding the Development of Risk-Averse Scientific Software. By D. Kelly. The Journal of Systems and Software. Volume 109. 2015. Pages 50-61.

[Kendrick, 2011] 101 Project Management Problems and How to Solve Them: Practical Advice for Handling Real-World Project Challenges. By T. Kendrick. AMACOM. 2011.

[Keyes, 2003] Software Engineering Handbook. By J. Keyes. Auerbach Publications. 2003.

[Kitchin, Dodge, 2011] Code/Space: Software and Everyday Life. By R. Kitchin, M. Dodge. The MIT Press. 2011.

[Kruchten, 2004] Putting the "Engineering" into "Software Engineering". By P. Kruchten. The Fifteenth Australian Software Engineering Conference (ASWEC 2004). Melbourne, Australia. April 13-16, 2004.

[Laplante, 2007] What Every Engineer Should Know About Software Engineering. By P. A. Laplante. CRC Press. 2007.

[Lea, 1994] Christopher Alexander: An Introduction for Object-Oriented Designers. By D. Lea. ACM SIGSOFT Software Engineering Notes. Volume 19. Issue 1. 1994. Pages 39-46.

[Leonhardt, 2000] John Tukey, 85, Statistician; Coined the Word 'Software'. By D. Leonhardt. The New York Times. July 28, 2000.

[Leveson, 2011] Applying Systems Thinking to Analyze and Learn from Events. By N. G. Leveson. Safety Science. Volume 49. Issue 1. 2011. Pages 55-64.

[Mahoney, 2004] Finding a History for Software Engineering. By M. S. Mahoney. IEEE Annals of the History of Computing. Volume 26. Number 1. 2004. Pages 8-19.

[Marques, Ochoa, 2013] Transferring Software Development Knowledge and Skills in the Academia: A Literature Review. By M. Marques, S. F. Ochoa. Technical Report

TR/DCC-2013-2. Computer Science Department. University of Chile. Santiago, Chile. 2013.

[McBreen, 2002] Software Craftsmanship: The New Imperative. By P. McBreen. Addison-Wesley. 2002.

[McConnell, 2004] Professional Software Development: Shorter Schedules, Better Projects, Superior Products, Enhanced Careers. By S. C. McConnell. Addison-Wesley. 2004.

[Miller, Voas, 2008] Computer Scientist, Software Engineer, or IT Professional: Which Do You Think You Are? By K. W. Miller, J. Voas. IT Professional. Volume 10. Issue 4. 2008. Pages 4-6.

[Monson-Haefel, 2009] 97 Things Every Software Architect Should Know: Collective Wisdom from the Experts. By R. Monson-Haefel (Editor). O'Reilly Media. 2009.

[Moor, 1985] What is Computer Ethics? By J. H. Moor. Metaphilosophy. Volume 16. Number 4. 1985. Pages 266-275.

[Muffatto, 2006] Open Source: A Multidisciplinary Approach. By M. Muffatto. Imperial College Press. 2006.

[Nerur, Balijepally, 2007] Theoretical Reflections on Agile Development Methodologies. By S. Nerur, V. G. Balijepally. Communications of the ACM. Volume 50. Issue 3. 2007. Pages 79-83.

[Offutt, 2013] Putting the Engineering into Software Engineering Education. By J. Offutt. IEEE Software. Volume 30. Number 1. 2013. Pages 94-96.

[O'Regan, 2013] Mathematics in Computing: An Accessible Guide to Historical, Foundational and Application Contexts. By G. O'Regan. Springer-Verlag. 2013.

[Parnas, 1998] Software Engineering Programmes are Not Computer Science Programmes. By D. L. Parnas. Annals of Software Engineering. Volume 6. Numbers 1-4. 1998. Pages 19-37.

[Parnas, 2011] Precise Documentation: The Key to Better Software. By D. L. Parnas. In: The Future of Software Engineering. S. Nanz (Editor). Springer-Verlag. 2011. Pages 125-151.

[Poppendieck, Poppendieck, 2007] Implementing Lean Software Development: From Concept to Cash. By M. Poppendieck, T. Poppendieck. Addison-Wesley. 2007.

[Rajlich, 2012] Software Engineering: The Current Practice. By V. Rajlich. CRC Press. 2012.

[Rich, Bateman, Esain, Massey, Samuel, 2007] Lean Evolution Lessons from the Workplace. By N. Rich, N. Bateman, A. Esain, L. Massey, D. Samuel. Cambridge University Press. 2007.

[Rost, Glass, 2011] The Dark Side of Software Engineering: Evil on Computing Projects. By J. Rost, R. L. Glass. John Wiley and Sons. 2011.

[Sangwan, Bass, Mullick, Paulish, Kazmeier, 2007] Global Software Development Handbook. By R. Sangwan, M. Bass, N. Mullick, D. J. Paulish, J. Kazmeier. Auerbach Publications. 2007.

[Sommerville, 2011] Software Engineering. By I. Sommerville. Ninth Edition. Addison-Wesley. 2011.

[Spinellis, 2015] Extending Our Field's Reach. By D. Spinellis. IEEE Software. Volume 32. Number 6. 2015. Pages 4-6.

[Spinellis, 2016] Developer, Debug Thyself. By D. Spinellis. IEEE Software. Volume 33. Number 1. 2016. Pages 3-5.

[Tedre, 2015] The Science of Computing: Shaping a Discipline. By M. Tedre. CRC Press. 2015.

[Tekinerdogan, Aksit, 2011] A Comparative Analysis of Software Engineering with Mature Engineering Disciplines Using a Problem-Solving Perspective. By B. Tekinerdogan, M. Aksit. In: Modern Software Engineering Concepts and Practices: Advanced Approaches. A. H. Doğru, V. Biçer (Editors). IGI Global. 2011. Pages 1-18.

[Tockey, 2004] Return on Software: Maximizing the Return on Your Software Investment. By S. Tockey. Prentice-Hall. 2004.

[Trendowicz, Jeffery, 2014] Software Project Effort Estimation: Foundations and Best Practice Guidelines for Success. By A. Trendowicz, R. Jeffery. Springer International Publishing. 2014.

[Wiegers, 2006] More About Software Requirements: Thorny Issues and Practical Advice. By K. E. Wiegers. Microsoft Press. 2006.

[Wohlin, Runeson, Höst, Ohlsson, Regnell, Wesslén, 2012] Experimentation in Software Engineering. By C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén. Springer-Verlag. 2012.