Presented at the 13th International Conference on Software & Systems Engineering and their Applications, December 5-8, 2000, CNAM - Paris, France

Systems Engineers are from Mars,

Software Engineers are from Venus[1,2]

Joseph Kasser
Systems Engineering and Evaluation Centre
University of South Australia,
Mawson Lakes Campus, Room F-37
Mawson Lakes, SA, 5095
Australia
Email: Joseph.Kasser@unisa.edu.au
Telephone +61 8 8302 3941 Fax +61 8 8302 5344

Sharon Shoshany
School of Computer and Information Science
University of South Australia,
Mawson Lakes Campus,
Mawson Lakes, SA, 5095
Australia
Email Sharon.Shoshany@unisa.edu.au

Abstract

In the last three decades of the 20th Century, a period roughly matching software engineering 's (SWE) history, the development of complex projects seems to have resulted in massive failures. This paper provides some insight that the failure of systems and software engineers to communicate, and more importantly their failure to understand that they are not communicating, may be a hitherto undetected cause of the failures of today's complex projects. The paper then explores some of the reasons for the communications failure and recommends an approach to bridging the gap.

Keywords

System engineering, software engineering.

[1] Acknowledgment is made to John Gray Ph.D. for the use of his metaphor.
[2] This work was partially funded from the DSTO SEEC Centre of Expertise Contract.

1.0 Introduction

Five thousand years of experience in hardware engineering have provided its practitioners with methodologies that have, in the main, been successful. It was during that time that projects were engineered. Only in the last three decades of the 20th Century, a period roughly matching SWE's history, has the development of complex projects been plagued by massive failures. Yet when seen from a historical perspective the complex projects of the last three decades are no more complex than the large engineering projects of the past, projects like the railroads, canals, pyramids and military sieges, within the constraints of the then-available tools and technology[3]. The growing recognition that the multidisciplinary environment in which today's complex software intensive systems are developed is characterized by poor requirements, poor change management and poorly defined processes resulted in several approaches to improve software development, including:

- The development of Computer Aided Software Engineering (CASE) Tools.
- The focus on process and methodology embodied in the ISO/IEEE standards.
- The United States of America's Department of Defense initiated Software Capability Maturity Model (CMM).
- The development of the "domain abstraction concept" in Object-Oriented Methodologies (OOM).
- The adoption of the Unified Modeling Language (UML).
- The development of reusable generic software components.
- The inclusion of software engineers on Integrated Product Teams (IPT).

However the adoption of these approaches will not guarantee that current and future software based complex projects will not fail. This is because the projects that software engineers develop do not exist in a vacuum [1]. Software cannot perform its function without an underlying hardware platform. Consequently, software engineers need to communicate with the systems engineers, hardware engineers and the other members of the IPT.

Realizing that a communications gulf existed, we tried to identify it. After some discussion[4] we developed a conceptual meta-model of the system development process within the system life cycle (SLC). The meta-model, which became our Rossetta Stone, summarizes the development process in the following manner:

When faced with the problem of meeting the customer's needs:

1. According to good engineering practice, there are two implementation choices:
   A. The problem is similar to other problems that have been solved in the past. Thus this time around, providing a similar solution may solve the problem. The process then becomes one of identifying the applicability of the solutions of the past, to the problem of the present and applying the elements of one or more solutions of the past to solve the problem of the present.
   B. The problem is unique so there are no known solutions. The process then becomes one of identifying a solution that makes the maximum use of existing solutions to past problems (components) and the minimum use of components to be developed so as to reduce the risk of non-delivery on time and within budget.
2. Engineers don't always reuse solutions or components that worked in the past, they reinvent them or try to invent new ones.

At this time, while we came to the realization that much of the communication problem was with the semantics, however we also identified other underlying barriers including:

- Training and Background Differences
- A lack of respect for the other's profession.
- The use of language
- The role of systems engineer in the SLC.
- Different Concepts.

Before considering each of the barriers to communications it is important to distinguish between a true barrier, and poor application of the methodology since each engineering discipline has both good and bad practitioners.

Each barrier in the above list is discussed in the paper and several examples are cited from the published literature as well as from personal experience

2.0 Training and Background Differences

This section analyzes the different training and background of Software, Systems, and hardware engineers.

[3] While we can see the products or outputs of those development processes, little information about cost and schedule overruns or prior failures has survived through the ages.

[4] No blood was shed in the process but we came close.

2.1 Hardware Engineers

The most common background for system engineers is hardware engineering. Moreover, hardware engineering is often used as a reference to describe generic processes. Therefore it is important to understand the background of the hardware engineer. Hardware engineers have been using models and blueprints in the form of schematic diagrams and sketches since the early days of engineering. They also use a component-based methodology. However, they develop components that are physical in nature, e.g. black boxes, printed circuit boards, integrated circuits and subsystems. When hardware engineers design a digital system, they use integrated circuits. They partition the design to make use of standard components and make use of a small amount of custom circuitry as required to interface the standard components. The engineer will pattern match sections of the design to the offerings in the vendor's family of components. The physical nature of the components themselves enforces the mapping of functionality onto physical components.

2.2 Software Engineers

In its early days software development practitioners used two basic approaches. Some developed their own libraries of subroutines or modules for functions, that once debugged, could be used in other programs. The majority however:

- Tried to implement the same functions in various ways.
- Suffered from the "not invented here" syndrome and would not reuse code from external sources.
- Failed to obtain the benefits of code reuse because they either changed working code or reused the code in a context different from its original one and the difference was the cause of the failure.

However, software engineers have since matured and created several more important methodologies, including

- Componentware - Components are large grained entities that are defined by their interfaces (services provided and demanded) and can be independently developed and used [2]. The component approach promises to turn software development to software assembly in a similar manner to the use of integrated circuits by the hardware engineer.
- Design Patterns – a way of expressing general solutions to recurring problems.
- Abstract modeling – a systematic way of capturing the system requirements in an abstract, consistent and complete manner.

Software engineers are often in the vanguard when it comes to system changes due to:

- The common misconception that it is easier to make software changes than hardware changes.
- Software is responsible in most cases for the user interfaces – an area constantly requiring changes.

Software engineers are often a product of a 3 years undergraduate course of computer and information science. This course seldom teaches math and physics in a level that is considered basic for engineering ("What do you mean you don't know what a Fourier Transform is" yelled the systems engineer at the software engineer.)

In a scan of the indices of books about SWE and development picked at random from those used for, or considered for use for, teaching SWE and management at the postgraduate level, only four books identified the term 'systems engineering' (SE) in their indices[5] [3,4,5,6]. The remainder made no mention of the term [7,8,9,10,11,12,13,14,15,16, 17,18,19,20,21,22,23]. Thus a generation of software engineers are seemingly being taught to engineer software without knowing much about traditional engineering and what function systems engineers perform. Thus SWE does not directly address SE and is also teaching methodologies for eliciting requirements (i.e. Use cases, business objects). At the same time it is ready to accept the system engineer as a requirement source in the manner of any other stakeholder.

2. 3 System Engineers

Systems engineers in general have little if any formal training in SE. They graduated to the discipline from another engineering discipline, mostly from hardware, therefore the software engineer claims that they may not have an understanding of the nature of software and software engineers are assuming the role of the systems engineer in software-intensive systems

The SE community has recognized the need for formal education and training in SE [24]. A scan of indices of books on SE for software had mixed results. Software is cited in the index of several books on SE [25,26,27,28,29,30,31]. Although the requirements for SWE are discussed in the context of adapting MIL-STD 2167A (SWE) to SE [32], software only shows up in the bibliography section of [29] and fails to show up completely in two books [33,34]. "Yes" pointed out the software engineer "while [34] doesn't list SWE in its index, it borrows a lot of terms and concepts from SWE".

[5] Two of the books are used in the management classes not the engineering classes..

3.0 The lack of mutual respect

IPTs containing engineers from multiple disciplines develop today's complex projects. One of the characteristics of successful teams is respect for each other's specialty knowledge. Yet systems and software engineers, in general tend to have little respect for each other. This section explores the reasons that system and software engineers fail to respect each other.

One reason deals with the discipline of engineering. Many of the software development personnel are not engineers but are programmers (equivalent to technicians). Other software engineers lack a formal background in math and physics which tends to preclude them from fully understanding a system that has more then just software in it. On the other hand, many hardware and systems engineers have some programming experience which makes them believe that they are "software experts". Comments heard in organizations included

- "I tried using a real time operating system 15 years ago and it didn't work, therefore it shall not be used in my project".
- "This is something that my high school kid can program in a week".

Some software engineers feel that systems engineers seldom provide the deliveries expected from them. The software engineer claimed that she knew of several organizations that had to retrain their system engineers when going through the CMM process in order to be able to qualify for the Level 2. Other common complaints were

- "They keep handing us new requirements on the way to the canteen".
- "They fail to uncover all the system behavior issues in their requirements, concentrating on the static and most apparent ones".

These examples of poor engineering lowered the respect for systems engineers in the eyes of the software engineers. On the other hand, from the systems engineer's perspective, Watts Humphrey created a CMM for the individual called the Personal Software Process (PSP) [15].

"His process, used by systems and hardware engineers for years, is being treated with wonder by the SWE community because of the reception that the PSP is reciving" said the system engineer.

"Please don't mention the PSP," she said, "The PSP never was an issue in my community and was never actually mentioned or used".

"But look at this" he said.

> "*These techniques (the CMM, PSP, and others like them) apply basic engineering and management principles to software. Over the years, we software practitioners convinced ourselves that software was different. The basics did not apply to us; we needed to break the mold. We were wrong. Software is different in some ways, but it has more in common with other fields than we want to admit. We must use what others have proven works at work*".[35]

In his eyes this quotation proves what SE has held for a long time, namely, SWE is one of many engineering disciplines.

The systems engineer then picked up a book on software reusability and read [36]

> *"In well-established disciplines like civil or electrical engineering, reuse is based on the existence of previously coded knowledge. There are two different levels of reuse to consider: (1) the reuse of ideas or knowledge and (2) the reuse of particular artifacts and components… Electrical engineers, for example, consult component catalogs, check which available part best fits the design constraint, and, in some cases, relax the original design requirements to take advantage of existing components*".

The systems engineer agrees with the author but thinks that it is poor engineering practice to relax the original design requirements without consulting the customer. He has seen it happen in the world of SE and knows that the correct approach, the relaxation of requirements takes the form of a sensitivity analysis and an informed decision with the customer. He reads on to discover

> *" We propose a model for reusability based on these observations and on the assumption that available components usually do not match the requirements perfectly, making adaptation the rule rather than the exception. Our approach is to provide an environment that helps locate components and that estimates the adaptation and conversion effort necessary for their reuse. The reuse process is as follows:*
>
> *A set of functional specifications is given. The user then searches a library of available components to find the candidates that satisfy the specifications.*
>
> *If a component that satisfies all the specifications is available, reusing it becomes trivial".*

This is good stuff he thinks, SWE has finally learnt to use components properly. He then reads on and discovers the following text:

> *"More typically, several candidates exist, each satisfying some specifications. We call them similar components. In this case, the problem becomes one of selecting and ranking the available candidates based on how well they match the requirements and on the effort required to adapt the nonmatching specifications.*
>
> *Once an ordered list of similar candidates is available, the reuser selects the easiest to reuse and adapts it".*

"Adaptation!" cries the systems engineer, "that's like performing the functional equivalent of drilling into an integrated circuit and attaching a wire internally, instead of designing some external circuitry to interface the wire to the component"

"No" replies the software engineer, "Adaptation does not necessarily mean changing or modifying the component. For example, when you bring a 110 Volt radio from the USA to Australia which uses 220 Volts how do you adapt it?"

"Well you can change the power supply internally, or add a transformer externally" was the reply.

"So the word adapt does not necessarily mean modify!" she said triumphantly.

"Yes" he admitted.

"So why did you immediately associate 'adapt' with 'modify'? She asked.

"Because in my experience, that's what software engineers did, years ago" he replied.

After some discussion we agreed that indeed it was the **perception that drove the reality**, and even if today's software developers did not modify existing components, the systems engineer's perception, based on his experience, shaped his respect for software engineers.

In addition systems engineers have the perception that SWE much the same as any other engineering discipline, is characterized by poor practice. For example, as the systems engineer said, [37] states

> *"The software drives system considerations such as performance and cost. For example, in a recent survey of 16 books on Object-Oriented design, only six had the word 'performance' in their index, and only two has the word 'cost'".*

"From the system engineering perspective, **requirements drive performance and cost as well as functionality, which in turn drive the software and all other parts of the system**. So why should [37] in the year 2000 have to point out cost and performance to SWE?" he asked.

She replied "You will not find cost in hardware engineering text book either, simply because the books discuss technology and not management. On the other hand I agree that in some areas of information systems technology performance is not treated with enough respect".

4.0 The use of language

Just as a common language opens the door to communication, so too the lack of it erects a barrier not easily overcome [38]. However, even with a common language, communications is not guaranteed. While the notion that Great Britain and the United States are separated by a common language [39] may be known, its ramifications are subtle and it is not an easy concept to understand unless one has been sensitized to it. An example of the situation occurred in the mid 1970's during contract negotiations between the United States based Communications Satellite Corporation and British Aerospace. The language of the meeting was English. The meeting became stuck on one point. Someone then suggested tabling the issue. Both sides agreed and the meeting deteriorated. The situation was much improved when the interpreter pointed out that the verb "to table" means:

- To place the subject **on top** of the table for **immediate** discussion - in English
- To place the subject **under** the table for **later** discussion - in American.

Consider the following SLC analog, concerning the meaning of the word "component".
The systems engineer quoted [34] p50

> *"a component of a system is a subset of the physical realization (and the physical architecture) of the system to which a subset of the system's functions have been (will be) allocated".*

They agreed that systems engineers live in the physical world, and the software engineer also stressed that components are not only physical but are also logical. So they decided to look up the UML 1.3 definition of a component, namely:

> *"a component is a physical, replaceable part of a system that packages implementation and*

> *provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files".*

"So components ARE physical" said the systems engineer, to which the reply came **"What's physical for us is logical for you**".

She explained further "SWE separates the issue of component (what it does) from both deployment (where it executes) and from its instances (how many times it runs). It creates another layer of abstraction that has more in it then just the physical entity as regarded by SE".

"I think I'm getting a headache" he sighed.

5.0 The role of systems engineer in the SLC

> *"Modern information technology products, even the software-intensive ones, are complex enough to require application of techniques from both disciplines. Accordingly, it becomes important to understand the relationships between relevant standards for systems and SWE".*[40]

ISO 12207 depicts the link between system and software as:

> *"This standard establishes a strong link between a system and its software. It is based upon the general principles of SE. The basic components of SE (e.g., analysis, design, fabrication, evaluation, testing, integration, manufacturing, and storage/distribution) form the foundation for SWE in the standard. This standard provides the minimum system context for software. Software is treated as an integral part of the total system and performs certain functions in that system. This is implemented by extracting the software requirements from the system requirements and design, producing the software, and integrating it into the system".*

While mentioning SE, the standard is silent as to the role of systems engineers. A search through engineering textbooks for the role of systems engineers showed that SE is defined in several ways

> *"An iterative process of top-down synthesis, development, and operation of a real-world system that satisfies, in a near optimal manner, range of requirements for the system"*[25].

> *"SE is the activity of specifying, designing, implementing, validating, installing and maintaining systems as a whole* [5].

> *"SE is an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem. SE integrates all the disciplines and specialty groups into a team effort forming a structured development process that proceeds from concept to production to operation"* [41].

These different perceptions of SE illustrate the point that systems engineers themselves have different perceptions as to their role in the SLC [42]. For example, they might direct or perform

- The high level design.
- Requirements and interface management.
- Activities that are not performed by other specialty disciplines.
- Inter-group coordination.
- The advocacy for the customer during the design and test phases of the task and whenever the customer is not present.

In addition, he said, "in the 20th Century paradigm, which is based on building hardware, systems engineers convert customer needs to system requirements", and quoted

> *Most of today's systems engineers really appear (work as) to be Requirements and Interface Engineers. They have the responsibility to validate the requirements since there's little point in building a system which conforms to requirements if the requirements are incorrect*"[42].

"In the 21st Century paradigm, the system engineer should lead the IPT". He added, to wich she pointed out, "some modern software-intensive systems can be developed effectively without systems engineers as long as the software engineers perform the requirements elicitation function".

To which he replied "[1] begins the process of describing a system by naming its parts and then identifying how the component parts are related to each other. The system is analyzed in terms of objects and activities. There is a role for requirements engineering to capture the system level requirements, but the book is silent as to how the requirements are allocated between the software and hardware subsystems".

6.0 The use of concepts

This section addresses the differences in the following few concepts

- Inheritance
- Blueprints
- Models
- Objects and classes
- Architecture
- Adaption
- Use of viewpoints

6.1 Inheritance

Systems and hardware engineers use inheritance in the form of physical and domain knowledge as shown in the following examples:

- When they build a printed circuit board they inherit mechanical aspects from other printed circuit boards. The board may also inherit connectors and interface circuits from previous boards.
- Spacecraft inherit environmental attributes i.e. thermal vacuum, thermal, and vibration.
- Aircraft inherit attributes to make them airworthy.
- Ships inherit attributes to stop them from sinking and protect them from the long term effects of sea water.

However they do not generally realize that they are employing the concept of inheritance, other then the obvious case of reuse, so they have no methodology for using the concept.

SWE uses inheritance as an integral part of Object Oriented techniques. In modern SWE you can inherit any classifier : interface, class, use case etc. Thus creating a powerful way of expressing and implementing ideas for specialization and generalization, accompanied by rules of refinement.

6.2 Blueprints

"We have always used Blueprints in the form of engineering drawings" he said.

"Software engineer use blueprints as a metaphor claiming that software is always is a blueprint even when implemented " she countered.

6.3 Models

"In my way a model is a representation of a product" he said. She replied "a model is a way of expressing knowledge in an abstract way, yet exact, without showing unnecessary details. We can use a model during analysis or design. We can use a model to generate implementation. Software engineers model as a way of communicating knowledge"

6.4 Objects and classes

In SWE

- An object is a discrete entity with a well-defined boundary and identity that encapsulates state and behavior. An object is a role-centered entity with internal data and a set of operations provided for that data. An Object is an instance of a class.
- A class is the type of an object. A class is a descriptor for a set of objects that share the same attributes, operations (methods), relationships and behavior. A class might capture real-world concept or design concept A class can inherit another class's operation, relationship and behavior either for expressing commonalties or as a way of making a more specific class. Objects can be instanciated from every class (not just from the more specialized one).

Objects and Classes originally were used in Object Oriented Programming (OOP) to achieve encapsulation, reuse, inheritance and abstraction, later migrated to Object Oriented Analysis (OOA) as a way of capturing the real world (the holistic approach).

In SE an object is a subsystem. There is no concept of class. "It goes back to the fact that you are actually not inheriting" she pointed out.

6.5 Architecture

In SWE an architecture is

> *"The organizational structure and associated behavior of a system. An architecture can be recursively decomposed into parts that interact through interfaces, relationships that connect parts, and constraints for assembling parts. Parts that interact through interfaces include classes, components and subsystems".* (UML)

or/and

> *"The set of design decisions about any system (or smaller component) that keeps its implementers and maintainers from exercising needless creativity*"[43].

In SE, the architecture of a system consists of the structure(s) of its parts (including design-time, test-time, and run-time hardware and software parts), the nature and relevant externally visible properties of those parts (modules with interfaces, hardware units, objects), and the relationships and constraints between them. There are a great many possibly interesting relationships.

"On the first reading they both look the same" she said, then added " the software is adding a layer of abstraction that allows the developer to extend his problem dimensions".

6.6 Adaptation

The concept of Adaptation was addressed above.

6.7 Use of viewpoints.

Systems engineers' views tend to be constrained in the physical realm. So they tend to mix the physical and abstract views [44]. This tends to result in a one-to-one mapping between the functional and the physical.

Software engineers may pick any number of abstract views and try to separate concerns by splitting them to different yet related viewpoint..

7.0 Discussion

There are several reasons for the gulf separating SWE from the hard engineering disciplines. A major barrier is the semantic barrier due to the different perceptions of the use of words. Both sides must be made aware that the situation is akin to Humpty Dumpty telling Alice **that when he uses a word it means just what he chooses it to mean — neither more nor less** [45]. Consequently, in an exchange of information, each side should use active listening techniques to minimize loss of meaning across their common interface.

Systems engineers tend to think physically, and move rapidly to solutions. They do this using the reductionalist approach of partitioning the big problem into a number of smaller problems on the assumption that if all the small problems are solved, then the big problem will also be solved. They also have a wide range of all sorts of components to assemble into their architectures (resistors, computers, tanks, aircraft, etc.)

Hardware components generally evolved from specific application to application Systems and hardware engineers tend to focus on solving the problem, if they have to build a component, it tends to be a special purpose component optimized for that application. Hardware engineers constructed their own computer cards until vendor components become available. Similarly, companies developed proprietary network protocols until standards were adopted.

Software engineers tend to think in abstractions and try to find an elegant solution to the problem (sometime staying abstract for a longer period than the system engineer feels comfortable with). In general, software does not have the benefit of thousands of years of component development. Thus when developing applications, software engineers may try to develop them in a manner that allows them be reused in the same or in other yet to be specified applications.

8.0 Conclusion

Looking for direct solution to a problem is a Martian characteristic while looking for solution by discussion (rather by abstraction in our case) is a Venusian characteristic [46], hence it can be truly stated that systems engineers are from Mars and software engineers are from Venus.

Authors

Joseph Kasser D.Sc. has been a practicing systems engineer for 30 years. He is the author of "*Applying Total Quality Management to Systems Engineering*" published by Artech House. He also developed the Master of Software Engineering Degree at University of Maryland University College, and teaches SWE at the postgraduate level via distance education. He participated in the review of versions 0.1 and 0.7 of the SEBOK [47]. British born, he lived in the United States of America long enough to become sensitized to the barriers to communications described in this paper. He is currently learning his third dialect of English, namely Australian.

Sharon Shoshany M.Sc. (Electrical Engineering) has been a software engineer for 16 years, developing and managing the development of large real time systems in the defense domain. Her background in electrical engineering enables her to attempt to cross the gap to SE or at least be able to appreciate its width.

References

[1] Pfleeger, S.L.: SWE Theory and Practice, Prentice Hall, New Jersey, 1998.
[2] Szyperski, C.: Component Software, Addison Wesley, 1997.
[3] Shumate, K., Keller, M.: Software Specification and Design A Disiplined Approach for Real-Time Systems, John Wiley & Sons Inc., 1992.
[4] Pressman, R.S.: A Manager's Guide to SWE, McGraw-Hill, 1993.
[5] Sommerville, I.: SWE, Addison-Wesley, Fifth Edition, 1998.
[6] Donaldson S.E.: Siegel, S.G., Cultivating Successful Software Development A Practitioner's View, Prentice Hall PTR, Upper Saddle River, New Jersey, 1997.
[7] Shlaer, S., Mellor, S.J.: OO Systems Analysis, Yourdon Press, 1988.
[8] Marcotty, M.: Software Implementation, Prentice Hall, 1991.
[9] Jacobson, I., Christerson, M., Jonsson, P.: _vergarrd, G., Object-Oriented SWE A Use Case Driven Approach, Addison Wesley, Revised 1993.

[10] Arthur, L.J.: Improving Software Quality An Insider's Guide to TQM, John Wiley & Sons, Inc., 1993.
[11] Yourdon, E.: Decline & Fall of the American Programmer, Yourdon Press, 1993.
[12] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[13] Perry, W.E.: Effective Methods for Software Testing, John Wiley & Sons, 2000.
[14] DeMarco, T.: Why Does Software Cost So Much? And Other Puzzles of the Information Age, Dorset House Publishing, 1995.
[15] Humphrey, W.: A Discipline for SWE, Addison-Wesley, Reading, Mass., 1995.
[16] Metzger, P.W., Boddie, J.: Managing a Programming Project Processes and People, Prentice Hall PTR, Upper Saddle River, New Jersey, 1996.
[17] BergJ, J-M., Levia, O., Rouillard, J.: Object-Oriented Modelling, Kluwer Academic Publishers, 1996.
[18] Yourdon, E., Argila, C.: OO Analysis and Design, Yourdon Press, 1993.
[19] Budd, T.: An Introduction to Object-Oriented Programming, 2nd Edition, Addison-Wesley, 1996.
[20] Fowler, M.: Analysis Patterns: Reusable Object Models, Addison-Wesley, 1997.
[21] Weinberg, G.M.: The Psychology of Computer Programming, Silver Anniversary Edition, Dorset House Publishing, 1998.
[22] Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, Addison Wesley, 1998.
[23] DeMarco, T., Lister, T.: Peopleware Productive Projects and Teams, Dorset House Publishing, 2nd Edition, 1999.
[24] Kasser, J.E.: “The Certified Systems Engineer: It’s About Time”, SETE 2000, Brisbane, Australia, 2000.
[25] Eisner H.: Computer Aided SE, Prentice Hall, Englewood Cliffs, NJ., 1988.
[26] Rechtin, E.: Systems Architecting Creating & Building Complex Systems, Prentice Hall PTR, Upper Saddle River, NJ., 1991.
[27] ThomJ, B., ed.: SE Principles and Practice of Computer-based SE, John Wiley & Sons, 1993.
[28] Eisner H.: Essentials of Project and SE Management, John Wiley & Sons, Inc., 1997.
[29] Martin, J.N.: SE Guidebook, CRC Press, 1997.
[30] Blanchard, B.: System Engineering Management, 2nd Edition, John Wiley & Sons, Inc., 1998.
[31] Kendall, K.E., Kendall, J.E.: Systems Analysis and Design, 4th Edition, Prentice Hall, Upper Saddle River, NJ, 1997.
[32] Kasser J.E.: Applying Total Quality Management to SE, Artech House, 1995.
[33] Hitchins, D.K.: Putting Systems to Work, John Wiley & Sons,1993.
[34] Buede, D.M.: The Engineering Design of Systems, John Wiley & Sons, Inc., 2000
[35] Phillips, D.: The Software Project Manager's Handbook Principles that Work at Work, IEEE Computer Society, Los Alamitos, California, 1998.
[36] Prieto-Díaz, Classification of Reusable Models, published in Software Reusability, ACM Press,1987, reprinted in Biggerstaff, T.J., Perlis, Ed., A.J.: Software Reusability Volume 1 Concepts and Models, ACM Press, 1989.
[37] Boehm, B.: "Unifying SWE and SE", Computer, March 2000.
[38] Cox C.: 1996, Welcoming Immigrants to a Diverse America: English as our Common Language of Mutual Understanding, http://policy.house.gov/documents/statements/englis h.htm, last accessed 10 August 2000.
[39] Shaw G.B.: "England and America: Contrasts," a conversation between Bernard Shaw and Archibald Henderson, Reader's Digest, February 1925, attrib.
[40] Moore, J.W.: SWE Standards A User's Road Map, IEEE Computer Society, Los Alamitos, California, 1998.
[41] INCOSE 2000.
[42] Kasser, J.E.: "SE: Myth or Reality", INCOSE 6th International Symposium, 1996.
[43] D'Souza, D.F., Willis, A.C.: 1998, Objects, Components, and Frameworks With Uml: The Catalysis Approach (Addison-Wesley Object Technology Series)
[44] Lykins H., Friedenthal S., Meilich A.: "Adapting UML for an OO SE Method (OOSEM)", INCOSE, 2000.
[45] Carroll, L, Through the Looking Glass, 1872
[46] Gray, J.: Men are from Mars, Women are from Venus, HarperCollinsPublishers, 1992.
[47] SEBOK2000, SWE Body of Knowledge Version 0.7, IEEE Computer Society, available at http://www.software engineerbok.org/, last accessed 10 August 2000.