Modern Software Engineering Concepts and Practices:
Advanced Approaches

Ali H. Doğru
Middle East Technical University, Turkey

Veli Biçer
FZI Research Center for Information Technology, Germany

Information Science REFERENCE

INFORMATION SCIENCE REFERENCE
Hershey · New York

Senior Editorial Director: Kristin Klinger
Director of Book Publications: Julia Mosemann
Editorial Director: Lindsay Johnston
Acquisitions Editor: Erika Carter
Development Editor: Joel Gamon
Production Coordinator: Jamie Snavely
Typesetters: Keith Glazewski & Natalie Pronio
Cover Design: Nick Newcomer

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Modern software engineering concepts and practices : advanced approaches / Ali H. Doğru and Veli Biçer, editors.
p. cm.
Includes bibliographical references and index.
Summary: "This book provides emerging theoretical approaches and their practices and includes case studies and real-world practices within a range of advanced approaches to reflect various perspectives in the discipline"-- Provided by publisher.
ISBN 978-1-60960-215-4 (hardcover) -- ISBN 978-1-60960-217-8 (ebook) 1. Software engineering. I. Doğru, Ali H., 1957- II. Biçer, Veli, 1980-
QA76.758.M62 2011
005.1--dc22
2010051808

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 1
A Comparative Analysis of Software Engineering with Mature Engineering Disciplines Using a Problem-Solving Perspective

Bedir Tekinerdogan
Bilkent University, Turkey

Mehmet Aksit
University of Twente, The Netherlands

ABSTRACT

Software engineering is compared with traditional engineering disciplines using a domain specific problem-solving model called Problem-Solving for Engineering Model (PSEM). The comparative analysis is performed both from a historical and contemporary view. The historical view provides lessons on the evolution of problem-solving and the maturity of an engineering discipline. The contemporary view provides the current state of engineering disciplines and shows to what extent software development can actually be categorized as an engineering discipline. The results from the comparative analysis show that like mature engineering, software engineering also seems to follow the same path of evolution of problem-solving concepts, but despite promising advances it has not reached yet the level of mature engineering yet. The comparative analysis offers the necessary guidelines for improving software engineering to become a professional mature engineering discipline.

INTRODUCTION

Since the early history of software development, there is an ongoing debate what the nature of software engineering is. It is assumed that finding the right answer to this question will help to cope with the *software crisis*, that is, software delivered too late, with low quality and over budget (Pressman, 2008; Sommerville, 2007). The underlying idea behind this quest is that a particular view

DOI: 10.4018/978-1-60960-215-4.ch001

on software development directly has an impact on the software process and artifacts. Several researchers fairly stated that in addition to the question what software development currently is, we should also investigate what professional software development should be. The latter question acknowledges that current practices can be unprofessional and awkward and might require more effort and time to maturate. Although both the questions on what software development is and what professional software development should be are crucial, it seems that there are still no definite answers yet and the debate is continuing from time to time after regular periods of silence. Some researchers might consider this just as an academic exercise. Yet, continuing the quest for a valid view of software development and a common agreement on this is important for a profound understanding, of the problems that we are facing with, and the steps that we need to take to enhance software development.

The significant problems we may face, though, seem not to be easily solved at the level as they are analyzed in current debates. To be able to provide both an appropriate answer to what software engineering is, and what it should be, we must shift to an even higher abstraction level than the usual traditional debates. This view should be generally recognized, easy to understand and to validate and as such provide an objective basis to identify the right conclusions. We think that adopting a problem solving perspective provides us an objective basis for our quest to have a profound understanding of software development. Problem-solving seems to be ubiquitous that it can be applied to almost any and if not, according to Karl Popper (2001), to all human activities, software development included. But what is problem-solving actually? What is the state of software development from a problem-solving perspective? What needs to be done to enhance it to a mature problem solving discipline? In order to reason about these questions and the degree of problem-solving in software development we have first to understand problem-solving better. Problem solving has been extensively studied in cognitive sciences such as (Newell et al., 1976; Smith et al., 1993; Rubinstein et al., 1980) and different models have been developed that mainly address the cognitive human problem solving activity. In this paper we provide the Problem Solving for Engineering Model (PSEM), which is a domain-specific problem solving model for engineering. This PSEM will be validated against the mature engineering disciplines such as civil engineering, electrical engineering and mechanical engineering. From literature (Ertas et al., 1996; Ghezzi et al., 1991; Wilcox et al., 1990; Shaw et al., 1990) it follows that engineering essentially aims to provide an engineering solution for a given problem, and as such, can be considered as a problem solving process. We could further state that mature engineering disciplines are generally successful in producing quality products and adopt likewise a mature problem-solving approach. Analyzing how mature engineering disciplines solve their problems might provide useful lessons for acquiring a better view on what software development is, that has not yet achieved a maturity level. Hence, we have carried out an in-depth comparative analysis of mature engineering with software engineering using the PSEM. In principle, every discipline can be said to have been immature in the beginning, and evolved later in time. Mature engineering disciplines have a relatively longer history than software engineering so that the various problem solving concepts have evolved and matured over a much longer time. Studying the history of these mature disciplines will justify the problem-solving model and allow deriving the concepts of value for current software engineering practices. Hence, our comparative study considers both the current state and the history of software development and mature engineering disciplines. Altogether, we think that this study is beneficial in at least from the following two perspectives. First, an analysis of software engineering from a problem-solving perspective will provide an

innovative and refreshing view on the current analysis and debates on software development. In some perspectives it might be complementary to existing analyses on software development, and in addition since problem-solving is at a higher abstraction level it might also highlight issues that were not identified or could not have been identified before due to the limitations of the adopted models for comparison. Second, the study on mature engineering disciplines will reveal the required lessons for making an engineering discipline mature. The historical analysis of mature engineering will show how these engineering disciplined have evolved. The analysis on the current practices in these mature engineering disciplines will show the latest success factors of mature engineering. We could apply these lessons to software engineering to enhance it to a mature problem solving, and thus a mature engineering discipline. In short, this study will help us to show what software development currently is, and what professional software development should be.

The remainder of this paper is organized as follows: The second section presents the problem-solving for engineering model (PSEM). The model defines the fundamental concepts of problem-solving and as such allows to explicitly reason about these concepts. In the third section, we use the PSEM to describe the history of mature engineering. The fourth section reflects on the history of software engineering based on the PSEM model and compares software engineering with mature engineering. In the fifth section, we provide a discussion and the comparison of software engineering with mature engineering. The sixth section presents the related work and finally the last section presents the conclusions.

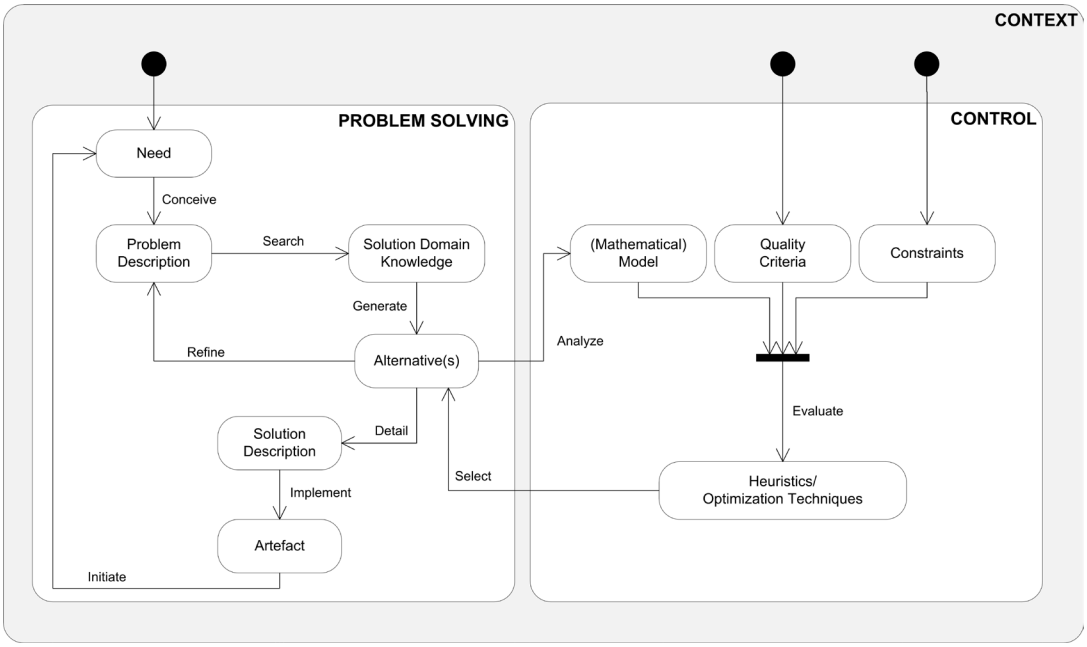
PROBLEM-SOLVING FOR ENGINEEING MODEL

Several survey papers (Deek et al., 1999; Rubinstein et al., 1980) represent a detailed analysis on the various problem-solving models. While there are many models of problem-solving, none has been explicitly developed to describe the overall process of engineering and/or compare engineering disciplines in particular. There have been problem-solving models for representing design as problem-solving (Braha et al., 1997), but no broad general model has been proposed yet which encompasses the overall engineering process.

A common model that represents engineering from a problem-solving will specifically show the important features of engineering. In this context, we could come up with a very abstract model for problem-solving consisting essentially of two concepts: *Need* and *Artifact*. Given a particular need (*Problem*) an artifact (*Solution*) must be provided that satisfies the need. Because of its very abstract nature, all engineering disciplines, including software engineering, apply to this overly simple model. Of course, the counterpart of the abstract nature of the model is that it is less useful in identifying the differences between the existing engineering disciplines and for comparing these. Hence, we are interested in a concrete problem-solving model that describes the separate important concepts needed for understanding and expressing the concepts of engineering. To this aim, we propose the domain specific *Problem-Solving for Engineering Model (PSEM)*, which is illustrated in Figure 1. In the subsequent sections, PSEM will serve as an objective basis for comparing engineering disciplines.

This domain specific model has been developed after a thorough literature study on both problem-solving and mature disciplines. In addition to the before mentioned problem-solving literature, we have studied selected handbooks including chemical engineering handbook (Perry, 1984), mechanical engineering handbook (Marks, 1987), electrical engineering handbook (Dorf, 1997) and civil engineering handbook (Chen, 1998). Further we have studied several textbooks on the corresponding engineering methodologies of mechanical engineering and civil engineering (Cross,

Figure 1. Problem-solving for engineering model (PSEM)



1989; Dunsheath, 1997; Shapiro, 1997), electrical engineering (Wilcox et al., 1990) and chemical engineering (Biegler, 1997).

The model is based on UML statecharts and consists of a set of states and transitions among these states. The states represent important concepts, the transitions represent the corresponding functions among these concepts. Concepts are represented by means of rounded rectangles, functions by directed arrows. The model consists of three fundamental parts: *Problem- Solving*, *Control* and *Context*. In the following, we will explain these parts in more detail.

Problem-Solving

The problem-solving part consists of six concepts: *Need*, *Problem Description*, *Solution Domain Knowledge*, *Alternative*, *Solution Description* and *Artifact*.

- *Need* represents an unsatisfied situation existing in the context. The function *Input* represents the cause of a need.
- *Problem Description* represents the description of the problem. The function *Conceive* is the process of understanding what the need is and expressing it in terms of the concept *Problem Description*.
- *Solution Domain Knowledge* represents the background information that is used to solve the problem. The function *Search* represents the process of finding the relevant background information that corresponds to the problem.
- *Alternative*, represents the possible alternative solutions. The function *Generate* serves for the generation of different alternatives from the solution domain knowledge. After alternatives have been generated, the problem description can be refined using the function *Refine*. The function *Detail* is used to detail the description of a selected alternative.

- *Solution Description* represents a feasible solution for the given problem.
- *Artifact* represents the solution for the given need. The function *Implement* maps the solution description to an artifact. The function *Output* represents the delivery and impact of the concept *Artifact* to the context. The function *Initiate* represents the cause of a new need because of the produced artifact.

Control

Problem-solving in engineering starts with the need and the goal is to arrive at an artifact by applying a sequence of actions. Since this may be a complex process, the concepts and functions that are applied are usually controlled. This is represented by the Control part in the model. A control system consists of a controlled system and a controller (Foerster, 1979). The controller observes variables from the controlled system, evaluates this against the criteria and constraints, produces the difference, and performs some control actions to meet the criteria. In PSEM, the control part consists of four concepts: *Representation of Concern, Criteria,* and *Adapter*.

- *(Mathematical) Model* represents a description of the concept Alternative. The function *Analyse* represents the process of analyzing the alternative.
- *(Quality) Criteria* represent the relevant criteria that need to be met for the final artifact. The function *Evaluate* assesses the alternative with respect to *(Quality) Criteria* and *Constraints*.
- *Constraints* represent the possible constraints either from the context or as described in *Problem* Statement.
- *Heuristics/Optimization Techniques* represents the information for finding the necessary actions to meet the criteria and constraints. The function *Select* selects the right alternative or optimizes a given alternative to meet the criteria and the constraints.

Context

Both the control and the problem-solving activities take place in a particular context, which is represented by the outer rounded rectangle in Figure 1. Context can be expressed as the environment in which engineering takes place including a broad set of external constraints that influence the final solution and the approach to the solution. Constraints are the standards, the rules, requirements, relations, conventions, and principles that define the context of engineering (Newell et al., 1976), that is, anything, which limit the final solution. Since constraints rule out alternative design solutions they direct engineer's action to what is doable and feasible. The context also defines the need, which is illustrated in Figure 1 by a directed arrow from the context to the need concept. Apparently, the context may be very wide and include different aspects like the engineer's experience and profession, culture, history, and environment (Rubinstein et al., 1980).

HISTORICAL PERSPECTIVE OF PROBLEM-SOLVING IN MATURE ENGINEERING

In the following, we will explain PSEM from an engineering perspective and show how the concepts and functions in the model have evolved in history in the various engineering disciplines. While describing the historical developments we will indicate the related concepts of PSEM in italic format in the corresponding sentences.

Directly Mapping Needs to Artifacts

Engineering deals with the production of artifacts for practical purposes. Production in the

early societies was basically done by hand and therefore they are also called craft-based societies (Jones et al., 1992). Thereby, usually craftsmen do not and often cannot, externalize their works in descriptive representations (Solution Description) and there is no prior activity of describing the solution like drawing or modeling before the production of the artifact. Further, these early practitioners had almost no knowledge of science (Solution Domain Knowledge), since there was no scientific knowledge established according to today's understandings. The production of the artifacts is basically controlled by tradition, which is characterized by myth, legends, rituals and taboos and therefore no adequate reasons for many of the engineering decisions can be given. The available knowledge related with the craft process was stored in the artifact itself and in the minds of the craftsman, which transmitted this to successors during apprenticeship. There was little innovation and the form of a craft product gradually evolved only after a process of trial and error, heavily relying on the previous version of the product. The form of the artifact was only changed to correct errors or to meet new requirements, that is, if it is necessary. To sum up, we can conclude that most of the concepts and functions of the problem-solving part in PSEM were implicit in the approach, that is, there was almost a direct mapping from the need to the artifact. Regarding the control part, the trial-and-error approach of the early engineers can be considered as a simple control action.

Separation of Solution Description from Artifacts

From history, we can derive that the engineering process matured gradually and became necessarily conscious with the changing context. It is hard to pinpoint the exact historical periods but over time, the size and the complexity of the artifacts exceeded the cognitive capacity of a single craftsman and it became very hard if not impossible to produce an artifact by a single person. Moreover, when many craftsmen were involved in the production, communication about the production process and the final artifact became important. A reflection on this process required a fundamental change in engineering problem-solving. This initiated, especially in architecture, the necessity for drafting or designing (Solution Description), whereby the artifact is represented through a drawing before the actual production. Through drafting, engineers could communicate about the production of the artifact, evaluate the artifact before production and use the drafting or design as a guide for production. This enlightened the complexity of the engineering problems substantially. Currently, drafting plays an important role in all engineering disciplines. At this phase of engineering, the concepts of Problem Description and Solution Description became explicit.

Development of Solution Domain Knowledge

Obviously classical engineers were restricted in their accomplishments when scientific knowledge was lacking. Over time, scientific knowledge gradually evolved while forming the basis for the introduction of new engineering disciplines. New advancements in physics and mathematics were made in the 17th century (Solution Domain Knowledge). Newton, for example, generalized the concept of force and formulated the concept of mass forming the basics of mechanical engineering. Evolved from algebra, arithmetic, and geometry, calculus was invented in the 17th century by Newton and Leibniz. Calculus concerns the study of such concepts as the rate of change of one variable quantity with respect to another and the identification of optimal values, which is fundamental for quality control and optimization in engineering. The vastly increased use of scientific principles to the solution of practical problems and the past experimental experiences increasingly resulted in the production of new

types of artifacts. The steam engine, developed in 1769, initiated the beginnings of the first Industrial Revolution that implied the transition from an agriculture-based economy to an industrial economy in Britain. In newly developed factories, products were produced in a faster and more efficient way and the production process became increasingly routine and specialized. In the 20th century the knowledge accumulation in various engineering disciplines has grown including disciplines such as biochemistry, quantum theory and relativity theory.

Development of Control Concepts and Automation

Besides of evolution of the concepts of the part Problem-Solving of Figure 1 one can also observe the evolution of the Control concepts. Primarily, mathematical modeling (Mathematical Model) seems to form a principal basis for engineering disciplines and its application can be traced back in various civilizations throughout the history. The development of mathematical modeling supported the control of the alternatives selection. Much later, this has led to automation, which is first applied in manufacture. The next step necessary in the development of automation was mechanization that includes the application of machines that duplicated the motions of the worker. The advantage of automation was directly observable in the increased production efficiency. Machines were built with automatic-control mechanisms that include a feedback control system providing the capacity for self-correction. Further, the advent of the computer has greatly supported the use of feedback control systems in manufacturing processes. In modern industrial societies, computers are used to support various engineering disciplines. Its broad application is in the support for drafting and manufacturing, that is, computer-aided design (CAD) and computer-aided manufacturing (CAM).

Contemporary Perspective of Problem-Solving in Mature Engineering

If we consider contemporary approaches in mature engineering then we can observe the following. First, the need concept in the PSEM plays a basic role and as such has directed the activities of engineering. In mature engineering, an explicit technical problem analysis phase is defined whereby the basic needs are mapped to the technical problems. Although initial client problems are ill-defined (Rittel, 1984) and may include many vague requirements, the mature engineering disciplines focus on a precise formulation of the objectives and a quantification of the quality criteria and the constraints, resulting in a more well-defined problem statement. The criteria and constraints are often expressed in mathematical formulas and equations. The quality concept is thus explicit in the problem description and refers to the variables and units defined by the International Systems of units (SI). From the given specification the engineers can easily calculate the feasibility of the end-product for which different alternatives are defined and, for example, their economical cost may be calculated.

Second, mature problem-solving also includes a rich base of extensive scientific knowledge that is utilized by a solution domain analysis phase (Arrango et al., 1994) to derive the fundamental solution abstractions. From our study it appears that each mature engineering is based on a rich scientific knowledge that has developed over several centuries. The corresponding knowledge has been compiled in several handbooks and manuals that describe numerous formulas that can be applied to solve engineering problems. The handbooks we studied contain a comprehensive coverage in-depth of the various aspects of the corresponding engineering field from contributions of dozens of top experts in the field. Using the handbook, the engineer is guided with hundreds of valuable tables, charts, illustrations, formulas, equations,

definitions, and appendices containing extensive conversion tables and usually sections covering mathematics. Obviously, scientific knowledge plays an important role in the degree of maturity of the corresponding engineering.

Third, in mature engineering different alternatives are explicitly searched from the solution domain and often organized with respect to predetermined quality criteria. Hereby, the quality concept plays an explicit role and the alternatives are selected in an explicit alternative space analysis process whereby mathematical optimization techniques such as calculus, linear programming and dynamic programming are adopted. In case no accurate formal expressions or off-the-shelf solutions can be found heuristic rules (Coyne et al., 1990; Cross et al., 1989) are used.

In mature engineering the three processes of technical problem analysis, solution domain analysis and alternative space analysis are integrated within the so-called *synthesis* process (Maimon et al., 1996; Tekinerdogan et al., 2006). In the synthesis process, the explicit problem analysis phase is followed by the search for alternatives in a solution domain that are selected based on explicit quality criteria.

In the synthesis process each alternative is analyzed through generally representing it by means of mathematical modeling. A mathematical model is an abstract description of the artifact using mathematical expressions of relevant natural laws. One mathematical model may represent many alternatives. In addition different mathematical models may be needed to represent various aspects of the same alternative. To select among the various alternatives and/or to optimize the same alternative Quality Criteria are used in the evaluation process that can be applied by means of heuristic rules and/or optimization techniques. Once the 'best' alternative has been chosen it will be further detailed (Detailed Solution Description) and finally implemented.

Summary

Reflecting on the history of mature engineering disciplines, we can conclude that the separate concepts of PSEM have evolved gradually. Traditional engineering disciplines such as electrical engineering, chemical engineering and mechanical engineering can be considered mature because the maturity of each concept in the PSEM.

Figure 2 shows the historical snapshots from the evolution of problem-solving in PSEM. In section 3.1, we have seen that problem-solving at the early phases of the corresponding engineering disciplines was rather simple and consisted of almost directly mapping needs to artifacts. In Figure 2, this is represented as time Ta. Later on, the concepts of Problem Description and Solution Description evolved (time Tb), followed by the evolutions of Solution Domain Knowledge and Alternatives (Tc), and finally the control concepts (Td) leading to PSEM as presented in Figure 1. Figure 2 is an example showing several snapshots. In essence, for every engineering discipline we could define the maturity degrees of the problem-solving concepts throughout the history.

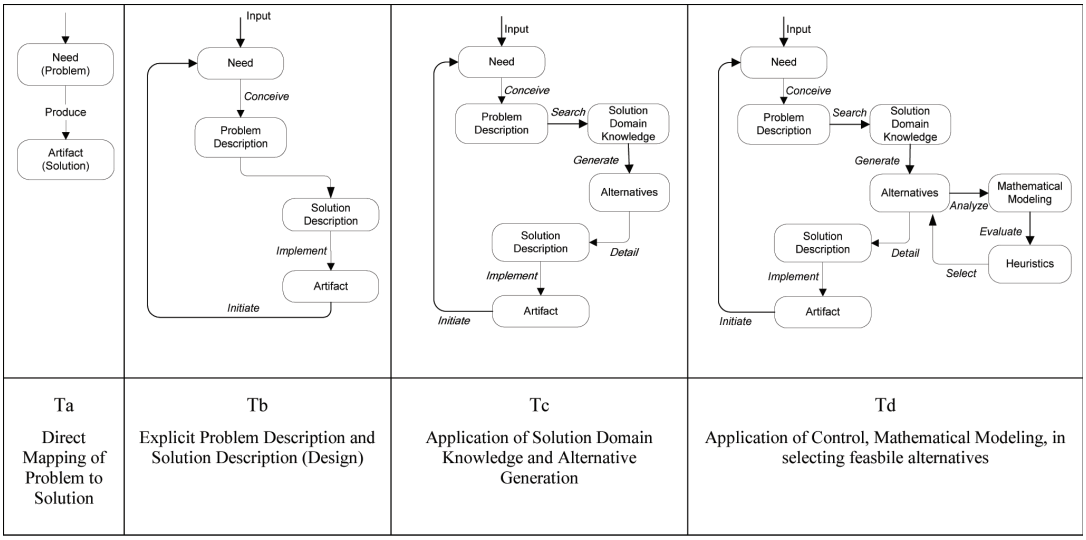
HISTORICAL PERSPECTIVE OF PROBLEM-SOLVING IN SOFTWARE ENGINEERING

We will now describe the historical development of problem-solving in software engineering. Although, the history of software engineering is relatively short and ranges only about a few decades, this study will illustrate the ongoing evolution of its concepts in PSEM and identify its current maturity level with respect to mature engineering disciplines.

Directly Mapping Needs to Programs

Looking back at the history we can assume that software development started with the introduction

Figure 2. Historical snapshots of the evolution of engineering problem-solving



Ta	Tb	Tc	Td
Direct Mapping of Problem to Solution	Explicit Problem Description and Solution Description (Design)	Application of Solution Domain Knowledge and Alternative Generation	Application of Control, Mathematical Modeling, in selecting feasbile alternatives

of the first generation computers in the 1940s such as the Z3 computer (1941), the Colossus computer (1943) and the Mark I (1945) computer (Bergin et al., 1996). The first programs were expressed in machine code and because each computer had its own specific set of machine language operations, the computer was difficult to program and limited in versatility and speed and size (Need). This problem was solved by assembly languages. Although there was a fundamental improvement over the previous situation, programming was still difficult. The first FORTRAN compiler released by IBM in 1957 (Bergin et al., 1996) set up the basic architecture of the compiler. The ALGOL compiler (1958) provided new concepts that remain today in procedural systems: symbol tables, stack evaluation and garbage collection (Solution Domain Knowledge). With the advent of the transistor (1948) and later on the IC (1958) and semiconductor technology the huge size, the energy-consumption as well as the price of the computers relative to computing power shrank tremendously (Context). The introduction of high-level programming languages made the computer more interesting for cost effective and productive business use. When the need for data processing applications in business was initiated (Need), COBOL (Common Business Oriented Language) was developed in 1960. In parallel with the growing range of complex problems the demand for manipulation of more kinds of data increased (Need). Later on the concept of abstract data types and object-oriented programming were introduced (Solution Domain Knowledge) and included in various programming languages such as Simula, Smalltalk, C++, Java and C#.

It appears that in the early years of computer science the basic needs did not change in variety and were directly mapped to programs. We can state that there was practically no design, no explicit solution domain knowledge and alternative analysis. In fact, this is similar to the early phases of mature engineering disciplines.

Separation of Solution Descriptions from Programs

The available programming languages that adopted algorithmic abstraction and decomposition have supported the introduction of many structured design methods (DeMarco, 1978; Jackson, 1975; Yourdon, 1979) during the 1970s, including differ-

ent design notations to cope with the complexity of the development of large software systems. At the start of the 1990s several object-oriented analysis and design methods were introduced (Booch, 1991; Coad et al., 1991) to fit the existing object-oriented language abstractions and new object-oriented notations were introduced. CASE tools were introduced in the mid 1980s to provide automated support for structured software development methods (Chikofsky, 1998). This had been made economically feasible through the development of graphically oriented computers. Inspired from architecture design (Alexander, 1977) design patterns (Gamma et al., 1995) have been introduced as a way to cope with recurring design problems in a systematic way. Software architectures (Shaw et al., 1996) have been introduced to approach software development from the overall system structure. The need for systematic industrialization (Need) of software development has led to component-based software development (Solution Description) that aims to produce software from pre-built components (Szyperski, 1998). With the increasing heterogeneity of software applications and the need for interoperability, standardization became an important topic. This has resulted in several industrial standards like CORBA, COM/OLE and SOM/OpenDoc. The Unified Modeling Language (UML) (Rumbaugh et al., 1998) has been introduced for standardization of object-oriented design models.

Development of Computer Science Knowledge

The software engineering community has observed an emerging development of the solution domain knowledge (Solution Domain Knowledge). Simultaneously with the developments of programming languages, a theoretical basis for these was developed by Noam Chomsky (1965) and others in the form of generative grammar models (Solution Domain Knowledge). Knuth presented a comprehensive overview of a wide variety of algorithms and the analysis of them (1967). Wirth introduced the concept of stepwise refinement (1971) of program construction and developed the teaching procedural language Pascal for this purpose. Dijkstra introduced the concept of structured programming (1969). Parnas (1972) addressed the concepts of information hiding and modules.

The software engineering body of knowledge has evolved in the last four decades (SWEBOK, 2004). This seems relatively short with respect to the scientific knowledge base of mature engineering. Nevertheless, there is now an increasing consensus that the body of knowledge is large and mature enough to support engineering activities. The IEEE Computer Society and the Association for Computing Machinery (ACM) have set up a joint project in which the so-called Software Engineering Body of Knowledge is developed (Bourque et al., 1999; SWEBOK, 2004) to characterize and organize the contents of the software engineering discipline.

Development of Control Concepts and Automation

The Control concepts have evolved in software engineering as well. Over the decades more and better case tools have been developed supporting software development activities ranging from architecture design to testing and software project management.

Mathematical modeling (Mathematical Model) and/or algebraic modeling is more and more integrated in software design. Empirical software engineering aims to devise experiments on software, in collecting data from the experiments, and in devising laws and theories from this data (Juristo et al., 2001). To analyze software systems, metrics are being developed and tested (Fenton et al., 1997).

Process improvement approaches such as, for example, the Capability Maturity Model Integration (CMMI) is proposed and applied (Boehm

et al., 2003). In parallel to these plan-based approaches agile software development has been advocated as an appropriate lightweight approach for high-speed and volatile software development (Boehm et al., 2003).

Currently the so-called model-driven software development (MDSD) aims to support the automation of software development (Stahl et al., 2006). Unlike conventional software development, models in MDSD do not constitute mere documentation but are considered executable similar to code. MDE aims to utilize domain-specific languages to create models that express application structure and behavior in a more efficient way. The models are then (semi)automatically transformed into executable code by model transformations.

The above developments are basically related to the enhancement of control in software engineering. Although this has not yet completed we can state that it follows similar path as in mature engineering.

Contemporary Perspective of Problem-Solving in Software Engineering

We have analyzed a selected set of textbooks on software engineering (Ghezzi et al., 2002; Pressman, 2004; Sommerville, 2007). In software engineering, the phase for conceiving the needs is referred to as requirements analysis, which usually is started through an initial requirement specification of the client. In mature engineering we have seen that the quality concept is already explicit in the problem description through the quantified objectives of the client. In software engineering this is quite different. In contrast to mature engineering disciplines, however, constraints and the requirements are usually not expressed in quantified terms. Rather the quality concern is mostly implicit in the problem statement and includes terms such as 'the system must be adaptable' or 'system must perform well' without having any means to specify the required degree of adaptability and/or the performance. Of course, the importance of requirements engineering has seriously changed over the last decade. There is an IEEE conference on Requirements Engineering, which has been running successfully since 1993, a Requirements Engineering journal, several serious textbooks on requirements engineering and a lot of research, which deals with both formalizing and measuring functional and non-functional requirements. Although we can observe substantial progress in this community it is generally acknowledged that the aimed state of mature engineering is unfortunately not reached yet.

A similar development can be observed for the organization and the use of knowledge for software engineering. The field of software engineering is only about 50 to 60 years old and obviously is not as mature as in the traditional engineering disciplines. The basic scientific knowledge, on which software engineering relies, is mainly computer science that has developed over the last decades. Progress is largely made in isolated parts, such as algorithms and abstract data types (Shaw, 1990; Shaw et al., 1996).

One of the interesting developments is the increasing size of pattern knowledge. The goal of patterns is to create a body of literature, similar to the mature engineering disciplines, to help software developers resolve common difficult problems encountered throughout all of software engineering and development. Several books have been written including many useful patterns to support to design and implementation. Nevertheless, if we relate the quantity of knowledge to the supporting knowledge of mature engineering disciplines, the available knowledge in software engineering is still quite meager. The available handbooks of software engineering (Ghezzi et al., 2002; Pressman, 2004; Sommerville, 2007) are still not comparable to the standard handbooks of mature engineering disciplines. Moreover, on many fundamental concepts in software engineering consensus among experts has still not been reached yet and research is ongoing.

In other engineering disciplines at phases when knowledge was lacking we observe that the basic attitude towards solving a problem was based on common sense, ingenuity and trial-and-error. In software engineering it turns out that this was not much different and the general idea was that requirements have to be specified using some representation and this should be refined along the software development process until the final software is delivered.

Regarding alternative space analysis we can state that the concept of Alternative(s), is not explicit in software engineering. The selection and evaluation of design alternatives in mature engineering disciplines is based on quantitative analysis through optimization theory of mathematics. This is not common practice in software engineering. No single method we have studied applies mathematical optimization techniques to generate and evaluate alternative solutions. Currently, the notion of quality in software engineering has more an informal basis. There is however, a broad agreement that quality should be taken into account when deriving solutions. As in other engineering disciplines, in software engineering the quality concept is closely related to measurement, which is concerned with capturing information about attributes of entities (Fenton et al., 1997).

DISCUSSION

Since the introduction of the term software engineering in 1968 the NATO Software Engineering Conference, there has been many debates on the question whether software development is an engineering discipline or not. We can identify different opinions in this perspective. Some authors view software engineering as a branch of traditional engineering often believe that concepts from traditional engineering need to apply to software development. For example, Parnas (1998) argued that software engineering is a "an element of the set, {Civil Engineering, Mechanical Engineering, Chemical Engineering, Electrical Engineering,....}." Others argue that software engineering is not an engineering discipline, but that it should be (McConnell, 2003). Again others claim that software is fundamentally different from other engineering artifacts and as such can and should not be considered as an engineering discipline.

Based on our historical analysis we argue that currently software engineering shows the characteristics of an engineering discipline, but has not evolved yet to the maturity level of the traditional engineering disciplines. If we would characterize the current state of software engineering based on Figure 2, then it would be somewhere between Tb and Tc. Obviously it is not possible to define the exact characterization in terms of crisp values simply because each concept in the PSEM might have a maturity degree of progress that cannot be expressed as yes or no. Table 1 presents an analytical overview in which the different properties of both software engineering and mature engineering are shown. The properties (left column) are derived from the PSEM. For each property, we have provided a short explanation derived from our analysis as described in the previous sections. Based on this we can identify the concrete differences of software engineering with mature engineering and are better able to pinpoint what needs more focus to increase the maturity level of software engineering.

In the coming years we expect that each of these concepts will further evolve towards a mature level. This can be observed if we consider the current trends in software development in which the concepts are developing in a relatively high pace. By looking at the concepts in Table 1 we can give several examples in this perspective.

For example, Michael Jackson (2000) provides in his work on so-called problem-frames an explicit notion of problem in requirements engineering. In the aspect-oriented software development community the notion of concern has been introduced and several approaches are proposed to identify, specify and compose concerns (Filman

Table 1. Comparison of mature engineering with software engineering

	Mature Engineering	Software Engineering
Technical Problem Analysis	Explicit problem description specified with quantified metrics. Well-defined problems.	Usually implicitly defined as part of the requirements and usually no quantification of required solution. Ill-defined problems.
Availability of Domain Knowledge	Very extensive solution domain knowledge compiled in different handbooks.	Basically knowledge for isolated domains in computer science. Increasing number of pattern catalogs
Application of Domain Knowledge	Explicit domain analysis process for deriving abstractions from solution domain.	Solution domain analysis not a common practice. In general applied in case reuse is required.
Solution Description	Rich set of notations for different problems.	Various design notations. Still lack of global standards.
Alternative Analysis	Explicit alternative space analysis; optimization techniques for defining the feasible alternatives	Implicit. Almost no systematic support for alternative space analysis.
Quality Measurement	Explicit quality concerns both for development and evaluation.	Quality is usually implicit. No systematic support for measuring quality in common software practices
Application of Heuristics	Explicitly specified in handbooks as a complementary means to mathematical techniques for defining feasible solutions.	Implicit in software development methods.

et al., 2004). In a sense, concerns can be viewed as similar to the notion of technical problem that we have defined in this paper.

The organization and modeling of domain knowledge has been addressed, for example, in SWEBOK (2004) and other work on taxonomies (Glass et al., 1995). In parallel with this we can see the increasing number of publication of different pattern catalogs for various phases of the software life cycle. Also we observe that textbooks on software engineering provide a broader and more in-depth analysis of software engineering and related concepts, which is reflected by the large size of the volumes.

The application of domain knowledge to derive the abstractions for software design is represented in the so-called domain analysis process that was first introduced in the reuse community and software product line engineering (Clements et al., 2002). Currently we see that it is also being gradually integrated in conventional software design methods, which are indicating on the use of domain-driven approaches (Evans, 2004).

Regarding design notations we can state that the software engineering community is facing a continuous evolution of design notations and the related tools (Budgen, 2003).

Alternative analysis is not really explicitly addressed but there are several trends that show directions towards this goal. In software product line engineering variability analysis is an important topic and the process for application engineering is applied to develop different alternative products from a reusable asset base (Clements et al., 2002). The case of quality measurement has been explicitly proposed in the work on software measurement and experimentation (Fenton et al., 1997).

RELATED WORK

Several publications have been written on software engineering and the software crisis. Very often software engineering is considered fundamentally different from traditional engineering and it is claimed that it has particular and inherent complexities that are not present in other traditional engineering disciplines. The common cited causes of the software crisis are the complexity of the problem domain, the changeability of software, the invisibility of software and the fact that software

does not wear out like physical artifacts (Booch, 1991; Budgen, 2003; Pressman, 2004). Most of these studies, however, lack to view software engineering from a broader perspective and do not attempt to derive lessons from other mature engineering disciplines.

We have applied the PSEM for describing problem-solving from a historical perspective. Several publications consider the history of computer science providing a useful factual overview of the main events in the history of computer science and software engineering. The paper from, for example, Shapiro (1997) provides a very nice historical overview of the different approaches in software engineering that have been adopted to solve the software crisis. Shapiro maintains that due to the inherently complex problem-solving process and the multifaceted nature of software problems from history it follows that a single approach could not fully satisfy the fundamental needs and a more pluralistic approach is rather required.

Some publications claim in accordance with the fundamental thesis of this paper that lessons of value can be derived from other mature engineering disciplines. Petroski (1992) claims that lessons learned from failures can substantially advance engineering. Baber (1997) compares the history of electrical engineering with the history of software engineering and thereby focuses on the failures in both engineering disciplines. According to Baber software development today is in a pre-mature phase analogous in many respects to the pre-mature phases of the now traditional engineering discipline that had also to cope with numerous failures. Baber states that the fundamental causes of the failures in software development today are the same as the causes of the failures in electrical engineering 100 years ago, that is, lack of scientific mathematical knowledge or the failure to apply whatever such basis may exist. This is in alignment with our conclusions. Shaw (1990) provides similar conclusions. She presents a model for the evolution of an engineering discipline, which she describes as follows: "Historically, engineering has emerged from ad hoc practice in two stages: First, management and production techniques enable routine production. Later, the problems of routine production stimulate the development of a supporting science; the mature science eventually merges with established practice to yield professional engineering practice". Using her model, she compares civil engineering and chemical engineering and concludes that these engineering disciplines have matured because of the supporting science that has evolved. Shaw distinguishes between craft, commercial and professional engineering processes. These distinct engineering states can be each expressed as a different instantiation of the PSEM. The immature craft engineering process will lack some of the concepts as described by the PSEM. The mature professional engineering process will include all the concepts of the PSEM.

Several authors criticize the lack of well-designed experiments for measurement-based assessment in software engineering (Fenton et al., 1997). They state that currently the evaluation of software engineering practices depend on opinions and speculations rather than on rigorous software-engineering experiments. To compare and improve software practices they argue that there is an urgent need for quantified measurement techniques as it is common in the traditional scientific methods. In the PSEM measurement and evaluation is represented by the control part. As we have described before, mature engineering disciplines have explicit control concepts. The lack of these concepts in software engineering indicates its immature level.

CONCLUSION

Software engineering is in essence a problem-solving process and to understand software engineering it is necessary to understand problem-solving. To grasp the essence of problem-solving

we have provided an in-depth analysis of the history of problem-solving in mature engineering and software engineering. This has enabled us to position the software engineering discipline and validate its maturity level. To explicitly reason about the various problem-solving concepts in engineering, in section 2 we have presented the Problem-solving for Engineering Model (PSEM) that uniquely integrates the concepts of problem-solving, control and context. It appears that mature engineering conforms to the PSEM and this maturation process has been justified by a conceptual analysis from a historical perspective.

The PSEM and the analysis have provided the framework and the context for the debates on whether software development should be considered as an engineering discipline or not. From our conceptual analysis we conclude that software engineering is still in a pre-mature engineering state. This is justified by the fact that it lacks several concepts that are necessary for effective problem-solving. More concretely, we have identified the three processes of technical problem analysis, solution domain analysis and alternative space analysis that are not yet complete and fully integrated in software development practices. Nevertheless, despite the differences between software engineering and mature engineering, one of the key issues in this analysis is that software development does follow the same evolution of the problem-solving concepts that can also be observed from the history of mature engineering disciplines. Although it has not yet achieved the state of a professional mature engineering discipline the consciousness on the required concepts is increasing. With respect to the developments in other engineering disciplines, our study shows even a higher pace of the evolution of problem-solving concepts in software engineering and we expect that it will approach mature engineering disciplines in the near future.

REFERENCES

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., & Angel, S. (1977). *A pattern language: Towns, buildings, construction*. New York: Oxford University Press.

Arrango, G. (1994). Domain analysis methods. In Schäfer, R., Prieto-Díaz, R., & Matsumoto, M. (Eds.), *Software reusability*. Ellis Horwood.

Baber, R.L. (1997). Comparison of electrical engineering of Heaviside's times and software engineering of our times. *IEEE Annals of the History of Computing archive, 19*(4), 5-17.

Bergin, T. J., & Gibson, R. G. (Eds.). (1996). *History of programming languages*. Addison-Wesley.

Biegler, L. T., Grossmann, I. E., & Westerberg, A. W. (1997). *Systematic methods of chemical process design*. Prentice Hall.

Boehm, B., & Turner, R. (2003). *Balancing agility and discipline*. Addison-Wesley.

Booch, G. (1991). *Object-oriented analysis and design, with applications*. Redwood City, CA: The Benjamin/Cummins Publishing Company.

Bourque, P., Dupuis, R., & Abran, A. (1999). The guide to the software engineering body of knowledge. *IEEE Software*, *16*(6), 35–44. doi:10.1109/52.805471

Braha, D., & Maimon, O. (1997). The design process: Properties, paradigms, and structure. *IEEE Transactions on Systems, Man, and Cybernetics*, *27*(2).

Brooks, F. (1975). *The mythical man-month*. Reading, MA: Addison-Wesley.

Budgen, D. (2003). *Software design* (2nd ed.). Addison-Wesley.

Chen, W. F. (1998). *The civil engineering handbook*. CRC Press.

Chikofsky, E. J. (1989). *Computer-Aided Software Engineering (CASE)*. Washington, D.C.: IEEE Computer Society.

Chomsky, N. (1965). *Aspects of the theory of syntax*. MIT Press.

Clements, P., & Northrop, L. (2002). *Software product lines: Practices and patterns*. Addison-Wesley.

Coad, P., & Yourdon, E. (1991). *Object-oriented design*. Yourdon Press.

Colburn, T. R. (2000). Philosophy of computer science, part 3. In *Philosophy and Computer Science* (pp. 127–210). Armonk, USA: M.E. Sharpe.

Coyne, R. D., Rosenman, M. A., Radford, A. D., Balachandran, M., & Gero, J. S. (1990). *Knowledge-based design systems*. Addison-Wesley.

Cross, N. (1989). *Engineering design methods*. Wiley & Sons.

Deek, F. P., Turoff, M., & McHugh, J. A. (1999). A common model for problem solving and program development. *IEEE Transactions on Education*, *4*, 331–336. doi:10.1109/13.804541

DeMarco, T. (1978). *Structured analysis and system specification*. Yourdon Inc.

Diaper, D. (Ed.). (1989). *Knowledge elicitation*. Chichester, UK: Ellis Horwood.

Dijkstra, E. W. (1969). *Structured programming, software engineering techniques*. Brussels: NATO Science Committee.

Dorf, R. C. (1997). *The electrical engineering handbook*. New York: Springer Verlag.

Dunsheath, P. (1997). *A history of electrical engineering*. London: Faber & Faber.

Ertas, A., & Jones, J. C. (1996). *The engineering design process*. Wiley.

Evans, E. (2004). *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley.

Fenton, N. E., & Phleeger, S. L. (1997). *Software metrics: A rigorous & practical approach*. PWS Publishing Company.

Filman, R.E. Elrad, T., Clark, S. & Aksit, M. (2004). *Aspect-oriented software development*. Pearson Eduction.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Reading, MA: Addison-Wesley.

Ghezzi, C., Jazayeri, M., & Mandrioli, D. (2002). *Fundamentals of software engineering*. Prentice-Hall.

Glass, R. L., & Vessey, I. (1995). Contemporary application domain taxonomies. *IEEE Software*, *12*(4), 63–76. doi:10.1109/52.391837

Jackson, M. (1975). *Principles of program design*. Academic Press. Jackson, M. (2000). *Problem frames: Analyzing and structuring software development problems*. Addison-Wesley.

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The unified software development process*. Addison-Wesley.

Jones, J. C. (1992). *Design methods: Seeds of human futures*. London: Wiley International.

Juristo, N., & Moreno, A. M. (2001). *Basics of software engineering experimentation*. Kluwer Academic Publishers.

Knuth, D. (1967). *The art of computer programming*. Addison-Wesley.

Knuth, D. (1974). Computer programming as an art. *Communications of the ACM, 17*(12), 667-673. Transcript of the 1974 Turing Award lecture.

Maimon, O., & Braha, D. (1996). On the complexity of the design synthesis problem. *IEEE Transactions on Systems, Man, and Cybernetics*, *26*(1).

Marks, L. S. (1987). *Marks' standard handbook for mechanical engineers*. McGraw-Hill.

McConnell, S. (2003). *Professional software development: Shorter schedules, better projects, superior products, enhanced careers*. Boston: Addison-Wesley.

Newell, N., & Simon, H. A. (1976). *Human problem solving*. Englewood Cliffs, NJ: Prentice-Hall.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, *15*(12). doi:10.1145/361598.361623

Parnas, D. L. (1998). Software engineering programmes are not computer science programmes. *Annals of Software Engineering*, 19–37. doi:10.1023/A:1018949113292

Perry, R. (1984). *Perry's chemical engineer's handbook*. New York: McGraw-Hill.

Petroski, H. (1992). *To engineer is human: The role of failure in successful design*. New York: Vintage Books.

Popper, K. (2001). *All life is problem solving*. Routledge.

Pressman, R. S. (2008). *Software engineering: A practitioner's approach*. McGraw-Hill.

Rapaport, B. (2006). *Philosophy of computer science: What I think it is, what I teach, & how I teach it. Herbert A. Simon Keynote Address*. NA-CAP Video.

Rittel, H. W., & Webber, M. M. (1984). Planning problems are wicked problems. *Policy Sciences*, *4*, 155–169. doi:10.1007/BF01405730

Rubinstein, M. F., & Pfeiffer, K. (1980). *Concepts in problem solving*. Englewood Cliffs, NJ: Prentice-Hall.

Rumbaugh, J., Jacobson, I., & Booch, G. (1998). *The unified modeling language reference manual*. Addision-Wesley.

Shapiro, S. (1997). Splitting the difference: The historical necessity of synthesis in software engineering. *IEEE Annals of the History of Computing*, *19*(1), 20–54. doi:10.1109/85.560729

Shaw, M. (1990). Prospects for an engineering discipline of software. *IEEE Software*, 15–24. doi:10.1109/52.60586

Shaw, M., & Garlan, D. (1996). *Software architecture: Perspectives on an emerging discipline*. Prentice Hall.

Smith, A. A., Hinton, E., & Lewis, R. W. (1983). *Civil engineering systems analysis and design*. Wiley & Sons.

Smith, G. F., & Browne, G. J. (1993). Conceptual foundations of design problem solving. *IEEE Transactions on Systems, Man, and Cybernetics*, *23*(5). doi:10.1109/21.260655

Sommerville, I. (2007). *Software engineering*. Addison-Wesley.

Stahl, T., & Völter, M. (2006). *Model-driven software development*. Wiley.

SWEBOK. (2004). *Guide to the software engineering body of knowledge*.

Szyperski, C. (1998). *Component software: Beyond object-oriented programming*. Addison-Wesley.

Tekinerdoğan, B., & Akşit, M. (2006). Introducing the concept of synthesis in the software architecture design process. *Journal of Integrated Design and Process Science*, *10*(1), 45–56.

Upton, N. (1975). *An illustrated history of civil engineering*. London: Heinemann.

von Foerster, F. (1979). Cybernetics of cybernetics. In Krippendorff, K. (Ed.), *Communication and control in society*. New York: Gordon and Breach.

Wilcox, A. D., Huelsman, L. P., Marshall, S. V., Philips, C. L., Rashid, M. H., & Roden, M. S. (1990). *Engineering design for electrical engineers*. Prentice-Hall.

Williams, M. R. (1997). *A history of computing technology*. IEEE Computer Society.

Wirth, N. (1971). Program development by stepwise refinement. *Communications of the ACM*, *14*(4), 221–227. doi:10.1145/362575.362577

Yourdon, E., & Constantine, L. L. (1979). *Structured design*. Prentice-Hall.