

On the Notion of Software Engineering: A Problem Solving Perspective

Bedir Tekinerdogan & Mehmet Aksit

**TRESE group
Department of Computer Science
University of Twente
P.O. Box 217 7500 AE Enschede, The Netherlands
e-mail: {bedir | aksit}@cs.utwente.nl
<http://trese.cs.utwente.nl>**

Abstract-- Despite of extensive efforts, software projects have to cope with the recurring problems of the software crisis. We argue that the software crisis problem is more deeply rooted than it is generally perceived and that the problem is in the first place conceptual rather than technical. This implies that software engineering as it is currently perceived and applied may lack some fundamental concepts that are necessary to produce software systems cost-effectively. This paper presents a broad and general view of software engineering in order to grasp its essence and identify the concepts that are necessary but are not well-defined or even missing. It has been shown that software engineering, mature engineering disciplines and philosophy all aim at solving problems in their own context. By using a common problem solving model, an in-depth comparative analysis of software engineering with mature engineering and philosophy is provided. This comparative analysis helps us to identify a number of issues of current software engineering practices.

Index terms--foundations of software engineering, problem-solving, philosophy, mature engineering, comparative analysis

1. Introduction

Advances in programming languages enabled the shift in focus from programming-in-the-small to programming-in-the-large. When software projects attempted to build large and complex software systems, they were confronted with software delivered over budget, late and with low quality. People soon realized that building large-scale complex software systems was fundamentally different from programming small systems. The term *software crisis* was invented at the NATO conference on Software Engineering in 1968 to characterize this software problem. More and more software has been written ever since the first program and this trend is ongoing. Today, most scientific and technological projects involve software, whether they are academic or industrial. Most of these projects, though, have still to cope with the large complexities of software development and fail to completely fulfill the promises of providing reliable, adaptable and easily maintainable software [Neumann 95][Gibbs 94].

We argue that the software crisis problem is more deeply rooted than it is generally perceived and that the problem is in the first place conceptual rather than technical. This implies that software engineering as it is currently perceived and applied may lack some fundamental concepts that are necessary to produce software systems cost-effectively. To improve the maturity level of software engineering it is required that the necessary concepts are identified and integrated in the software engineering paradigm.

A problem cannot be solved at the same level as it was initiated. To grasp the missing concepts a broad and general view of software engineering is needed. Abstracting from software engineering we can firstly identify that it is a specialization of engineering. If we consider the various definitions and attributed meanings of engineering in various engineering literatures, it follows that engineering is in essence a problem solving process in which an engineering solution is sought for a given problem [Ertas & Jones 96][Ghezzi et al. 91][Wilcox et al. 90] [Shaw 90][Cross 89]. In software engineering, for example, the problem is stated by the requirement specification and the software engineer needs to provide a software solution. From the many studies on problem solving we can derive that problem solving is not particular to engineering but is generally applied [Hunt 94][Smith & Browne 93][Rubinstein & Pfeiffer 80][Newell & Simon 76]. A fundamental discipline that applies problem solving is philosophy. The primary goal of philosophy is to understand the nature of things and as such attempts to identify and describe essential concepts [Kolenda 74]. Engineering disciplines are intrinsically related to philosophy because many engineering concepts are derived from philosophy.

From the above we deduce that for a thorough understanding of software engineering it is required that we understand the essence of problem solving and engineering and comprehend the related fundamental philosophical concepts.

The novelty of this paper is that it presents a broad and general view of software engineering in order to grasp its essence and identify the concepts that are necessary but are not well-defined or even missing. For this, an in depth comparative analysis of software engineering with mature engineering and philosophy is provided based on problem solving concepts. Because of the adopted broad view, in addition to the software engineers, the paper may be of value for engineers of other disciplines and philosophers as well. Engineers of other disciplines may identify the explicit concepts of engineering and analyze, position and validate their own corresponding engineering discipline. Philosophers may identify the relation between engineering, science and philosophy and further reflect on this matter. The outline of this paper is presented in Figure 1.

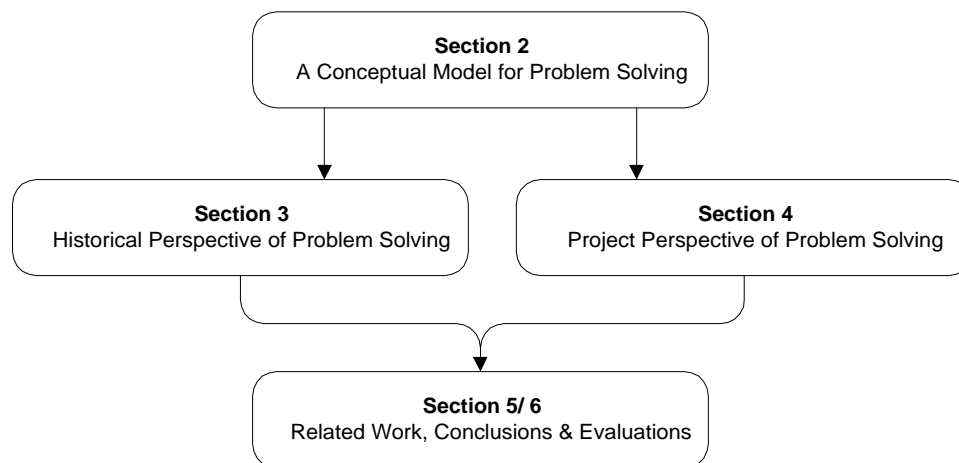


Figure 1. The outline of the paper

In section 2 a conceptual model for problem solving is presented. The model defines the fundamental concepts of problem solving and as such allows to explicitly reason about these. The model is very scaleable and it can be applied both for explaining the historical evolution of a discipline and for describing a particular project, that is, a plan that need to be executed within a given period of time. The historical perspective will be described in 3, the project perspective in section 4. These sections can be studied rather independently from each other.

In section 3 we will use the problem solving model to describe the history of philosophy, mature engineering and software engineering. Mature engineering disciplines and philosophy have a relatively longer history than software engineering in which the various problem solving concepts have evolved and matured over time. Studying the history of these disciplines will justify the problem solving model and allow to derive the concepts of value for current software engineering practices. A historical overview of software engineering is necessary to understand the evolution and the state of the art of software engineering and as such identify and validate the software engineering concepts with respect to the problem solving concepts.

In section 4 we will analyze problem solving in mature engineering and software engineering from project perspective. For this purpose, we will propose a conceptual model for engineering that is

derived from the previous problem solving model. This model will be used to identify the basic concepts of contemporary mature engineering disciplines, which may provide useful lessons for current software engineering practices. Section 4.1 will describe mature engineering from a project perspective. In section 4.2, we will analyze software engineering from a project perspective to understand the current software engineering processes and consequently compare these with the mature engineering practices. The comparative analysis will be based on the conceptual model of engineering. From this comparison study we will derive the deficiencies of the current software engineering paradigm. This will enable us to describe the gap between pre-mature software engineering and mature software engineering.

In section 5 we will discuss the related work on problem solving and comparative analysis studies.

Finally, in section 6 we will give the evaluations and conclusions that include the fundamental conceptual problems of software engineering. Based on the comparative analysis we will present the necessary requirements to solve these problems.

2. A Conceptual Model for Problem Solving

2.1 The CPC Model

In this section we propose a conceptual model of problem solving, which is illustrated in Figure 2. The model consists of a set of concepts and functions, which are represented by means of, rounded rectangles and directed arrows, respectively. Concepts are the necessary fundamental abstractions and the functions are the conceptual processes that describe the interactions between these concepts.

This is a controlled problem solving process, which takes place in a certain context. Therefore, we term this model as the *Controlled Problem Solving in Context* model, or the *CPC* model for short. Based on this assumption the model consists of three parts: *Problem Solving*, *Control* and *Context*. Below, we will explain these parts in more detail.

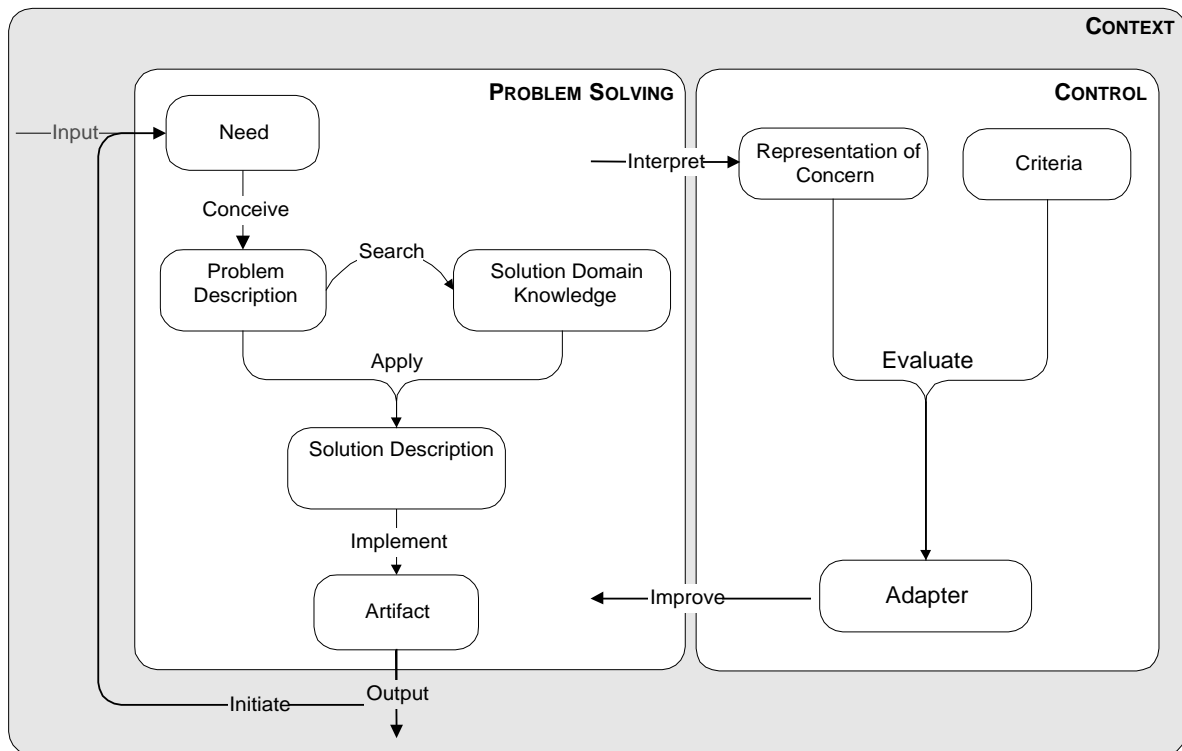


Figure 2. The Controlled Problem solving in Context Model (CPC Model)

2.1.1 Problem Solving

The problem-solving part consists of 5 concepts: *Need*, *Problem Description*, *Solution Domain Knowledge*, *Solution Description* and *Artifact*.

The function *Input* represents the cause of a need.

The concept *Need* represents an unsatisfied situation existing in the context.

The concept *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*.

The concept *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.

The concept *Solution Description* represents a feasible solution for the given problem.

The function *Apply* requires two inputs, *Problem Description* and *Solution Domain Knowledge*. It uses the relevant background information to provide a solution description that conforms to the problem description.

The concept *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 as a result of the produced artifact.

2.1.2 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 it is sometimes necessary to be controlled and improved. Therefore, we think that the concepts and functions of the controlling process must be modeled as well. A control system consists of a *controlled system* and a *controller* [Foerster 79]. 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 criteria¹. In our model we suggest that the control part consists of three concepts: *Representation of Concern*, *Criteria* and *Adapter*.

The function *Interpret* represents the process of retrieving information from the concept or function that needs to be controlled.

The concept *Representation of Concern* represents a description of the concept or function that is controlled.

The concept *Criteria* represents the relevant criteria that need to be met for the given concept or function.

The function *Evaluate* computes the difference between the actual state of the concept or function and the desired state of the concept or function. It provides the difference to the concept *Adapter*.

The concept *Adapter* represents the information for finding the necessary actions to meet the criteria.

The function *Improve* performs the required actions to meet the criteria.

The functions *Interpret* and *Improve* represent the link between *Problem Solving* and *Control* and can be in principle linked to any concept or function. This is to say that control may be applied to any concept or function of problem solving.

¹ This confirms to the view of cybernetics, which emphasizes mechanisms that allow complex systems to maintain, adapt, and self-organize [Umplebey 90].

2.1.3 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 2. 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 rules, requirements, relations, conventions, and principles that define the context of engineering [Newell & Simon 76], that is, anything, which limits the final solution. Since constraints rule out alternative design solutions they direct engineers action to what is doable and feasible.

The context also defines the need, which is illustrated in Figure 2 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 [Smith & Browne 93].

2.2 Purpose of the CPC Model

This CPC model serves as a basis for the whole paper. The purpose of this model is as follows:

First, we would like to gain a general understanding of problem solving. The CPC model defines the fundamental concerns of problem solving and abstracts from problem solving processes in philosophy and engineering. In this way we aim to better understand and describe each concept individually. For example, we may describe the concept of *Need* for different engineering disciplines like mechanical engineering, electrical engineering and software engineering in a more general way.

Second, we would like to understand the process of problem solving. Each problem solving process follows a common functional pattern, which has been made explicit by the CPC model. This allows us to describe the functions individually. For example, we may describe the function *Conceive* in the conceptual model from distinct engineering perspectives.

Third, since each engineering discipline can be considered as an instantiation of the CPC model we can use it to analyze mature engineering disciplines and compare these with the more immature software engineering discipline. This comparative analysis may help us to identify the missing concepts/processes of software engineering.

Fourth, we intend to evaluate current software practices using this model. Since we are able to discuss about individual concepts and functions in the model, we may use the conceptual model for engineering as a reference model to analyze and describe the different software engineering practices. For example, we will consider the object-oriented software development paradigm with respect to this model.

Fifth, the comparison and evaluation of software engineering may provide us opportunities to identify the fundamental problems of software engineering and its practices. The problems of software engineering may be detected if the model can not be clearly represented in practices.

The following sections are structured around the above purposes of the model. Section 3 will mainly address the first two issues, that is, understanding the problem solving concepts and functions. Section 4 will discuss the third and the fourth issues, that is, comparing software engineering practices with the mature engineering practices and the evaluation of the different software engineering practices. Finally, section 5 will address the last issue, that is, the conclusions and evaluations.

3. Historical Perspective of Problem Solving

We aim to gain a broad understanding of the concepts adopted in engineering and improve our consciousness about it. To this aim we will present the historical perspective of problem solving that will provide a survey of the evolution of the problem solving concepts in history. The motivation for this is that from history we can observe the reason and the process in which way the concepts in a field has been developed and matured. As a matter of fact, a reflection on the experiences and knowledge in the past will also increase our consciousness about problem solving.

Although problem solving is a general process that is applied in a wide range of disciplines we need to select the relevant disciplines that are somehow related to software engineering. In this paper we have chosen to study the historical perspective of problem solving in philosophy, mature engineering and software engineering.

In section 3.1 we will present an overview of the history of problem solving in philosophy. Philosophy is the rational and critical study of concepts for the purpose of arranging concepts into a unified system and to improve the consciousness about these concepts. The history of philosophy extends over a period of more than two thousand years. Studying the history of philosophy may accordingly provide us a deeper understanding of the concepts of problem solving. In addition we may identify fundamental concepts in philosophy that may be of value or necessary to be included in the software engineering field.

In section 3.2 we will present an overview of the history of problem solving in mature engineering including mechanical engineering, electrical engineering, chemical engineering and civil engineering. As we described before, engineering is in essence a problem solving process. Problem solving in mature engineering disciplines have developed and matured over a period that ranges from several centuries to several thousands years. Like in the case of philosophy, studying the problem solving approaches of these mature engineering disciplines is therefore useful to identify how the concepts of the CPC model have developed over time. In addition since software engineering is a specialization of engineering the history of mature engineering may contain valuable lessons to software engineering.

Finally, in section 3.3 we will describe the history of problem solving in software engineering. The history of software engineering is relatively short and ranges only about a few decades. The history of

software engineering may show how the concepts in the CPC model have evolved for it and as such allow to identify its current maturity level.

3.1 Historical Perspective of Problem Solving in Philosophy

In the following we will explain the CPC model from the history of philosophy [Melchert 95][Kolenda 74] and likewise attempt to clarify the corresponding concepts and functions. For this reason, where appropriate we will refer to the concepts and functions of the CPC model between parenthesis.

3.1.1 From Mythology to Rational Problem Solving

It is generally agreed that Western philosophy started in the ancient Greek in the 6th century BC when early democracy was established and the economy and culture flourished, leaving room for a critical thought on the nature of things (*Context*) [Melchert 95]. Unsatisfied with the existing mythological explanations² the first philosophers sought for more rational answers to their basic questions on the natural phenomena (*Input*). Their basic concern was to find the essence, a primary substance, from which all things are originated (*Problem Description*). Their professions were often astronomer, mathematician and physician (*Solution Domain Knowledge*)³. They approached this problem by adhering to direct observations of the nature and critical thought (*Apply*). This system of thought in which the mythical explanations were abandoned can be considered as a first step towards scientific thought. The question on the primary substance was answered in various ways (*Solution Description*). For some of the philosophers the primary substance was a directly observable element in the nature. Thales thought that this basic element was water; Anaximenes, argued that this was air; Heraclitus believed that this was fire; Empedocles maintained that all things are composed of four elements: air, water, earth, and fire. Other philosophers attributed this primary substance to more abstract elements. For example, Anaximander maintained that this irreducible substance is the indeterminate *apeiron*. Pythagoras, concluded that the *number* is the essence of reality. Democritus believed that nature was constituted of an infinite number of *atoms*, invisible elements differing only in form, weight and size. All of these philosophies⁴ took for granted that objective truth existed that could be discovered through a critical exchange of ideas by a community of thinkers.

From the above we have shown that the early history of philosophy conforms to the problem solving concept as defined in the CPC model. The control concepts can be explained in the following way. In

² Homer's book "Iliad and the Odyssey", provides two major epics of ancient Greek on the many Gods to which the cause of various natural phenomena were attributed.

³ The term "philosopher" was only later introduced by Heraclitus

⁴ The original writings of the early philosophers no longer exist, but have been articulated in the works of Aristotle.

general, the philosophers reflected (*Interpret, Representation of Concern*) on the problem solving process of the philosophical treatments and attempted to improve this. What is reflected on, how it is reflected on, and in what way the process and/or functions are accordingly changed depends on the person who is attempting to interpret and improve the problem solving process. In the above context each philosopher commented on the writings (*Artifact*) of contemporary philosophers and tried to improve this with new theories. Many of these philosophers were also disciplines of earlier philosophers. Each philosopher had its own specific belief and value system (*Criteria*). Their evaluation (*Evaluate*) of the existing philosophical treatments therefore resulted in different proposals (*Adapter*) and extended the available knowledge (*Improve*).

It appears that this process of controlled problem solving within a context is applicable for the rest of the history of philosophy. In the following we will describe the control concept only for radical improvements of the philosophical problem solving process.

3.1.2 From Subjectivity to Conceptualization of Objective Knowledge

The treatments of the first philosophers were rationally based but their explanations were still speculations since the scientific justification by experimentation was lacking. In addition, there were many theories describing the natural phenomena each on their own different way, that is, there was not an objective view. This divergence of views was also observed in the moral life, when the Greek got contact with other populations adopting different customs and value of morality and justice (*Context*). These observations led (*Initiate*) the people to an inconvenience (*Need*) and determined the basis for a crisis in Greek life at the end of the fifth century BC. A movement called *Sophism* realized that this need was to be satisfied (*Conceive, Problem Description*). The Sophists⁵ were teachers of various subjects like rhetoric, dialectic grammar and logic (*Solution Domain Knowledge*). The Sophists reasoned that knowledge is essentially empirical and relative to man (*Apply*). According to the Sophists, the principle of morality is just that what satisfies one's instinct and passions, there is no better way to live. Derived from the need to explain the right moral conduct, the writings of the Sophists had their potential application in practical life (*Output*).

Socrates could not accept the view of the Sophists that objective truth does not exist and likewise an objective basis for moral life is missing (*Initiate, Need*). Socrates argued (*Search*) that although the knowledge of man is partial and not certain there should be ideas that are self-evident, necessary and accepted by all men, that is, he introduced the notion of *concept* (*Solution Description*). He affirmed the existence of concepts in the field of logic and morality. According to Socrates, perfect knowledge consists in understanding through concepts and these concepts can be attained by critical thinking

⁵ One of the most famous sophists was Protagoras (485-410 B.C.), the author of the statement "Man is the measure of all things".

and collaborative reasoning. Socrates lived what he thought by openly discussing with people in the street and as such educating them the critical thinking process (*Output*)⁶.

Plato continued (*Initiate, Input*) the treatment on concepts and developed his theory on *Ideas*, which describe the nature of the concepts (*Solution Description*). He contributed to the introduction of the notion of abstraction and classification, and discussed the fundamental problems of natural science. Aristotle systematized the basic concepts of many theoretical sciences such as physics, mathematics, art, biology, ethics and politics. In addition he provided rules for analyzing basic concepts and correct reasoning with the available knowledge.

We can consider the contributions of Socrates, Plato and Aristotle as a fundamental change in the problem solving process in philosophy. Their interpretation (*Interpret, Model Representation*) of the writings (*Artifact*) of earlier philosophers together with their idealistic attitude (*Criteria*) led to the change (*Improve*) of the approach for knowledge acquisition, knowledge representation and knowledge interpretation. The conceptualization and representation of knowledge formed a significant basis for the scientific developments in later centuries (*Output*).

3.1.3 From Solution Description to Implementation

With Aristotle the Greek political and social life broke down; Greece was involved in wars and first dominated by Persians and later became a province of the Roman Empire (146 BC) (*Context*). The loss of freedom and the destruction caused by the wars resulted (*Initiate*) in social problems (*Input*). Philosophy at this time mainly addressed the need for a proper ethical life (*Need*). Different philosophical approaches such as *Epicureanism*, *Skepticism* and *Stoicism* (*Solution Description*) advocated specific life attitudes to solve these problems. *Eclecticism* suggested combining the good of all systems. *Neoplatonism* founded by *Plotinus* was a religious philosophy based on the works of Plato and had a great influence on medieval thought (*Output*).

After the 3rd century Christianity entered the Greek world and had spread over the Roman Empire (*Context*). Philosophy had now turned its attention from scientific investigation to a philosophical understanding of religious questions (*Input, Need*). Toward the end of the 4th century, *Augustine* developed a system of thought that formed a synthesis of some of the elements of Platonic philosophy with the essentials of Christianity (*Solution Description*).

⁶ Socrates was later arrested for heresy and corrupting the young people. Finally, he was convicted and sentenced to death.

3.1.4 Preserving and Development of Solution Domain Knowledge

At this phase we observe a fundamental change in the context of problem solving in philosophy. During the Middle-Ages, philosophy and the quest for knowledge and truth further developed in the Muslim world that spanned Persia, Spain and North Africa (*Context*). They had founded universities and preserved both the ancient texts and classical learning to a great degree⁷ (*Solution Domain Knowledge*). Centers, such as the *House of Wisdom* in Baghdad, were founded by the ruling caliphs for translation and study of Greek and Indian scientific and philosophical works. Most of the contemporary philosophers tried to unify science, religion and philosophy (*Need*). Next to philosophy and Islamic religion many of these philosophers studied therefore natural sciences such as mathematics, physics, medical science, chemistry and astronomy (*Solution Domain Knowledge*). At the beginning of the eighth century, the neoplatonic philosopher Al-Kindi tried to work out an appropriate synthesis of philosophy with theology affirming the foundations for a monotheistic religion. Al-Farabi drew on the work of Plato and Aristotle to create a universal Islamic philosophy and attempted to systematize human knowledge with his monumental work, *Catalogue of Sciences* (*Solution Domain Knowledge*). Ibn Sina, known as Avicenna in the West, translated a collection of treatises on Aristotelian logic, metaphysics, psychology and the natural sciences. He contributed to medical science with his famous book *al-Qanun*, an encyclopaedia of medicine, which surveyed the entire medical knowledge available from ancient and contemporary sources. Ibn Rushd, known as Averroes in the West, wrote extensive analyzes of Aristotelean works⁸ and his philosophical work on meta-physics greatly influenced the philosophy in medieval Europe.

The developments of problem solving in the muslim world led to a change in context (*Context*) of problem solving in philosophy in Europe. This change may be considered similar to that of the Muslim world who came earlier in contact with the philosophical and scientific writings of the ancient Greek, China and India.

3.1.5 Philosophical Approaches for Deriving Knowledge

The translated and advanced philosophical and scientific treatments were now gradually made available to Europe from the 11th century (*Output*). First, through the Arabs who had conquered the southern border of the Mediterranean, later by the Turks who captured Constantinople, now Istanbul, in 1453. It was the contact with Greek science (*Context*), which laid the basis (*Input*) for the Renaissance movement in the 14th till the 16th century, which saw a renewed interest (*Need*) in classical thought and the arts. The introduction of paper from the Muslim world (which had acquired it from

⁷ In 900 AD in Spain, the University of Cordoba, for example, had 600.000 titles in its library. The University of Toledo had 400.000 titles.

⁸ For this reason, in the West he became also known as "The Commentator".

China) and the invention of the movable metal type by Gutenberg revolutionized the production of European books in the 15th century and, in part as a result of the Protestant Reformation, increased public literacy (*Solution Domain Knowledge*).

The contact with classical Greek thought together with the commentaries of Ibn Rushd and other Islamic scholars gave rise to new philosophical schools. From the 11th to the 15th century, Western thought was dominated by the philosophy of scholasticism, which attempted to use philosophical reasoning to understand Christian revelation. To preserve the integrity and supremacy of Roman Catholic doctrine (*Need*) Thomas Aquinas formed a synthesis of Christianity and the philosophy of Aristotle in the 13th century. This doctrine became later the official philosophy of the Roman Catholic Church (*Output*). The renewed interest in classical thought and the movement of scholasticism (*Context*) resulted (*Initiate*) in philosophical thought systems that addressed topics beyond theology. Among the basic philosophical movements were Rationalism and Empiricism, which provided different approaches (*Solution Description*) for obtaining knowledge. Rationalism developed by Descartes, Spinoza and Leibniz, emphasized that knowledge and truth can be deduced by reason from basic definitions and axioms. Spinoza even organized his work in Euclidian geometrical form including definitions, axioms, propositions and deductive proofs. In contrast to Rationalism, Empiricism, developed in Great Britain by Bacon, Hobbes, Locke, Berkeley and Hume, emphasized the importance of induction from sense experiences to obtain knowledge. Rationalism and Empiricism provided two opposite views for obtaining and judging knowledge (*Initiate*). To solve this problem (*Problem Description*), Kant synthesized Rationalism and Empiricism in his philosophy (*Solution Description*). According to Kant, all knowledge starts with experience, but it is the human mind that arranges knowledge by its own nature. Kant argued that although the human reason can construct science it is not able to construct metaphysics. Kant's philosophy plays a fundamental role (*Output*) in subsequent philosophical treatments.

3.1.6 Control of Problem Solving and Hermeneutics Philosophy

Interpretation plays an essential role in controlling the problem solving process. In philosophy, interpretation concerns mainly that of the philosophical writings. We will now focus on this function *Interpret*.

During the Renaissance movement (*Context*) the need to reconstruct the original texts of classical thought and interpret these put a focus on the principles of interpretation (*Initiate*). This problem was also identified in the realm of religion where the authority to interpret scriptures was a basic concern (*Need, Problem Description*). Earlier, in the Muslim world the caliphs possessed no authority to interpret the Quran and recordings of the tradition of Mohammed but the interpretation was established through a consensus of Muslim scholars. In Christianity, Luther, who initiated the Reformation movement, maintained that the interpretation of the Bible was a matter of individual

study, which could be done without regard to the contemporary authoritative Church doctrine⁹. The authority of the interpretation of texts was thus broadened. Another problem was *how* the classical texts should be interpreted. Up until then, the main assumption was that a text could be interpreted from its structural form and external referents of the text. In contrast to this, hermeneutic philosophy, argued that any formal syntax, will fail to completely determine its own interpretation and should be rather grounded on the original meanings of the author and their relevance for the authors (*Solution Description*). Hermeneutics is based on the philosophies of Heidegger, Dilthey, Gadamer, Vygotsky and Foucault among others.

In the second half of the 19th century Positivism arose as a system of philosophy which further emphasized experience and empirical knowledge of natural phenomena, in which metaphysics and theology are regarded as inadequate and imperfect systems of knowledge. During the early 20th century the Logical Positivism movement founded by Wittgenstein, Russel and Moore, who were concerned with developments in modern science, rejected the metaphysical doctrines of the traditional positivism and emphasized that knowledge should be scientifically verifiable.

3.2 Historical Perspective of Problem Solving in Mature Engineering

In the following we will explain the CPC model from an engineering perspective and show how the concepts and functions in the model have evolved in history in the various engineering disciplines. For this, we studied the history of the traditional engineering disciplines [Upton 75][Partington 70][Burstall 63][Dunsheath 62][Forbes 58]. We will mainly focus on the mature engineering disciplines as civil engineering, mechanical engineering, chemical engineering and electrical engineering.

3.2.1 Directly Mapping Needs to Artifacts

Engineering deals with the production of artifacts for practical purposes [Krick 69]. In fact, the words *engineering*, *engine* and *ingenious* are derived from the same Latin root, *ingenerare*, which conformably means *to create*.

To meet the various human needs man has always put effort in the creation of devices that solve their practical problems and make natural resources more useful. The basic concerns of man in ancient times were shelter, food gathering, agriculture, domestication of animals and hunting (*Need*). To support these needs the principal engineering activity included making (*Implement*) houses, tools and weapons (*Artifact*).

⁹ Later, this was followed by a fragmentation of the Christianity into various religious groups.

To increase force and use it more efficiently (*Need*) artifacts like the lever, the pulley and the wheel (*Artifact*) were already produced before 3000 BC. A particular application of the lever was the balance beam for weighing, which can be considered as the beginnings of measurement and experimentation in engineering [Burstall 63].

Stone was the basic material for the production of the early artifacts. For example, in ancient Egypt, to preserve and protect the bodies of the pharaohs for eternity, giant pyramids including temples and tombs were built out of stone. The engineering method included a great supply of human labor and only the elementary mechanical principles were applied.

The advent of metal, first of copper and bronze and later of iron, improved the quality of the tools drastically. At first the native metals such as gold or copper which sometimes occur in nature in a pure state were used, but later metallurgy developed when man learned how to smelt metallic ores by heating them to obtain the metals (*Solution Domain Knowledge*).

The rise of cities led to specialization and the division of labor. In villages and nomadic societies most of the people were directly involved in food production, whereas in cities also other professions became important, like smith, trader or priest. Cities increased the rise of commerce and industry, architecture, art and learning, and as such they played an essential role in the emergence of all great civilizations. As cities in the early civilizations increased in size and density of population (*Context*), communication with other regions became necessary for food supply and other commerce (*Need*). For this reason, roads¹⁰ and bridges were built (*Artifact*). To sustain plant growth, and thus the food production, irrigation was needed in places where rainfall does not provide enough moisture. For this reason canals, basins and dams were produced.

Production in the early societies was basically done by hand and therefore they are also called craft-based societies [Jones 92]. 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 [Alexander 64]. 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

¹⁰ During the Roman Empire, for example, 300.000 kilometers of roads were built, among which the famous still remaining *Via Appia*, the primary road from Rome to Greece.

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 really necessary.

To sum up, the process of problem solving in engineering was simply based on practical know-how, common sense, ingenuity and trial and error. In a sense, there was thus little consciousness about the engineering activities, which is the reason why Alexander terms such engineering processes as *self-unconscious*¹¹ [Alexander 64]. Due to this unconsciousness we can conclude that most of the concepts and functions of the problem solving part in the CPC model 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.

3.2.2 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. Over time the size and the complexity of the artifacts exceeded the cognitive capacity¹² 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 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.

3.2.3 Development of Mathematical Solution Domain Knowledge

In addition to the separation of the product description from the product itself, the knowledge increased also gradually. It is clear that in early engineering problem solving the concepts and functions of the CPC model were implicit and as such were not distinguishable as separate concepts.

¹¹ According to Alexander the fast reaction to single failures and the resistance to other needless changes made the self-unconscious process self-adjusting, which is one of the fundamental problems in modern engineering.

¹² Experiments from psychology suggest that the maximum number of meaningful chunks of information an individual person simultaneously can comprehend is on the order of seven plus or minus two [Miller 56]. In addition the processing speed of the human mind is limited as well: it takes the mind about 5 seconds to accept a new chunk of information [Simon 62].

There is a clear relation with the maturity of the solution domain knowledge and the maturity of engineering. We will now therefore look at the concept *Solution Knowledge* in more detail.

One of the basic knowledge that later emerged and is applied in engineering is mathematics. In ancient Egypt, Sumer and Babylonia empirical geometry was adopted for land surveying and architecture (*Need*) [MSEncarta 96]. This was later refined and systematized by the Greeks. In the 6th century BC Pythagoras laid the basics of scientific geometry by showing that the various arbitrary and unconnected laws of empirical geometry could be derived from a basic set of *axioms*. Later, Euclid organized the Greek geometry of the time in his famous book, *Elements*.

Although, early Greek science was merely a generalization from experiences and had a speculative character, they provided, inspired from philosophical thought, the notions of concept and abstraction mechanisms to express relations between apparently disconnected phenomena [Hull 59]. This provided one of the fundamental tools for knowledge modeling and subsequent scientific inquiry (*Solution Domain Knowledge*).

The Greek arithmetic was mainly theoretical and was not suitable for rapid calculations in practice (*Need*), which is generally attributed to an insufficient numerical notation (*Problem Description*). After the Greek, mathematics was further developed in the Islamic world. In the 9th century, Al-Khawarizmi¹³ helped to introduce the Arabic numerals, the decimal position system and the concept of zero to arithmetic leading to a substantial improvement in calculations in contemporary engineering (*Solution Domain Knowledge*). In addition he introduced the concept of algorithm, which provided a universal method for solving a problem by repeatedly using a simpler computational method. This formed the basis for formalization of methods in engineering. Arithmetic dealt with only specific instances of mathematical relations. Al-Khawarizmi, introduced the notion of algebra, which generalizes mathematical relations such as the Pythagorean theorem and as such formed the conceptual language for mathematics.

3.2.4 Development of Solution Domain Knowledge through Experimentation

The improvement in scientific knowledge formed later the basis for the introduction of new engineering disciplines and the development of existing ones. Chemistry, the basis of modern chemical engineering, evolved also from the ancient period. The writings of some of the early Greek philosophers about the fundamental substance of the universe might be considered to contain the first chemical theories. In ancient Egypt and China, Aristotle's theory that all things tend to reach perfection formed the fundamental concept of alchemy. Because other metals were thought to be less perfect than gold, it was reasonable to assume that gold was gradually formed out of other metals within the earth. If this process could be artificially carried out in the workshop, gold would be

¹³ The word "algorithm" is derived from his name.

gained to increase or to prolong life, as it was believed in China. Although, based on incorrect theories, alchemy provided useful practical chemistry knowledge and played an important role in other scientific developments.

Together with the further development of scientific knowledge a focus on empirical experimentation (*Apply*) started in the Muslim world, which developed laboratories and workshops [Garrison 91] for this purpose. Combined with the developed mathematical knowledge and the other scientific knowledge different artifacts were produced for various purposes. For example, they developed astrolabe for measuring the positions of heavenly bodies and the pendulum that is used in several kinds of mechanical devices. In chemistry, the processes of evaporation, filtration, sublimation, melting, distillation and crystallization were developed and alcohol¹⁴, sulfuric and nitric acids and gasoline were produced [Al-Hassan & Hill 86]. These processes are described in Al-Razi's *Book of Secrets*, which also gives a full account of equipment for these chemical-processing techniques. To control the use of water, which was precious in these countries, the Muslim engineers produced extensive hydraulic systems, such as water-raising machines.

3.2.5 Development of Scientific Solution Domain Knowledge

Obviously, classical engineers were restricted in their accomplishments when scientific knowledge was lacking. Only after the scientific knowledge was broadened new types of artifacts could be produced for solving practical problems.

The contact with the Muslims after the 11th century made the accumulated technology and knowledge also available to Europe. The Scientific Revolution in Europe started with Copernicus who proposed in 1543 a heliocentric model of the universe, in which the sun is at the center of the universe and the planets move in concentric circles around it. Copernicus' heliocentric theories could explain and predict more astronomical facts than the geo-centered model of Ptolemy, which had been adopted since the 3rd century. However, his calculations of astronomical positions were not decisively accurate and mostly based on speculation. In this sense, although revolutionary in content, it was not so in method [Hull 59]. Later, Galileo introduced a scientific approach based on systematic measurements through planned experiments, rather than speculation (*Apply*). New types of artifacts were produced in this period also, for example, tools like the telescope, microscope and the thermometer which on their turn all supported the experimental scientific methods. Galileo, for example, made use of the telescope in observation and the discovery of sunspots, lunar mountains and valleys, the four largest satellites of Jupiter, and the phases of Venus. Based on his experimentation he discovered the laws of falling bodies and the motions of projectiles.

¹⁴ The word "alcohol" is derived from the Arabic word "Al-Kuhul" which denotes kohl, a fine powder. In medieval Europe this was applied to essences obtained from distillation, which led to its current use.

New advancements in physics and mathematics were made in the 17th century (*Solution Domain Knowledge*). Newton 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. In 1642 the philosopher and mathematician Pascal devised the first calculating adding machine, a precursor of the digital computer. In 1670s Leibniz devised a machine that could also multiply. Pascal formulated in conjunction with Fermat the mathematical theory of probability, which has become important as a fundamental element in the calculations of modern theoretical physics.

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 Industrial Revolution that implied the transition from an agriculture-based economy to an industrial economy. In newly developed factories, products were produced in a faster and more efficient way and the production process became increasingly routine and specialized.

3.2.6 Specialization of Problem Solving Techniques

Chemical engineering and chemistry advanced in the 19th century. Through the development of electrochemistry and spectroscopy many more chemical elements could be discovered. Mendeleev and Meyer independently developed the chemical law that states that the properties of all the elements are periodic functions of their atomic weights. In 1869 Mendeleev proposed *the Periodic Table of Elements* that classifies the chemical elements corresponding to their atomic weights. Based on this table subsequent discoveries of new elements were made which led to the completion of the table. In the 19th century chemical engineering witnessed an enormous advance in polymer technology and in the 20th century the mass production of polymers became economically feasible. These advances led to the introduction of new material, such as, plastics and fibers.

The basis for electrical engineering were founded in the 19th century and extended in the 20th century. Faraday discovered electromagnetic induction and the laws of electrolysis and deduced the principle of the generator, induction coil and transformer [Garrison 91]. James Clerk Maxwell laid out the theory of electromagnetic waves in a series of papers published in the 1860s. He analyzed mathematically the theory of electromagnetic fields and predicted that visible light was an electromagnetic phenomenon.

In the 20th century the knowledge accumulation in various engineering disciplines has grown. In chemistry, biochemistry was founded, which has unraveled the genetic code and explained the function of the gene. Quantum theory and relativity theory formed the basis for subsequent physics.

Heisenberg's uncertainty principle, formulated in 1927, had a substantial role in the development of quantum mechanics and also in the trend of modern philosophical thinking. The theory states that it is impossible to specify simultaneously the position and momentum of a particle with precision. In quantum mechanics, probability calculations therefore replace the exact calculations of classical mechanics.

In modern industrial societies computer is used to support various engineering disciplines. The most broad application is its use for drafting and manufacturing, that is, computer-aided design (CAD) and computer-aided manufacturing.

3.3 Historical Perspective of Problem Solving in Software Engineering

We will now describe the historical development of problem solving in software engineering. The survey is not purely chronological but is described from the perspective of the CPC model.

3.3.1 Directly Mapping Needs to Programs

Looking back at the history we can assume that software development started with the introduction of the first generation computers in the 1940s such as the Z3 computer (1941), the Colossus computer (1943) and the Mark I (1945) computer. These computers were basically used for numerical calculations for military purposes during the second world-war (*Need*) [Palfreman & Swade 93]. These numerical problems were directly 'programmed' (*Implement*) on the computers by setting switches and plugging cables into sockets. Programming was made easier through the improved architecture for computers defined by a paper of Von Neumann in which all of the basic elements of a stored-program computer were presented (*Solution Domain Knowledge*). The stored program concept meant that instructions to run a computer for a specific function, known as a *program*, were held inside the computer's memory, and could quickly be replaced by a different set of instructions for a different function.

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, which replaced the cryptic binary codes for the computer operations with symbolic notations. Later on, this process was automated by means of assembler programs. The first assembler was introduced in 1954 by IBM [Williams 97]. Although there was a fundamental improvement over the previous situation, programming was still difficult.

The first FORTRAN (Formula Translation) compiler was released by IBM in 1957 [Bergin & Gibson 96]. Similar to the Von Neumann architecture for computers, this compiler set up the basic architecture of the compiler. In ALGOL (Algorithm Language) (1958) new concepts were added that remain today in procedural systems: symbol tables, stack evaluation and garbage collection (*Solution*

Domain Knowledge). LISP (LISt Processor), implemented by McCarty at MIT in 1958 was a language designed for symbolic processing and formed the basis for the functional software programming paradigm. Intended for artificial intelligence programming its earliest applications included programs that performed symbolic differentiation, integration, and mathematical theorem verification.

It appears that in the early years of computer science the basic needs did not change in variety and were directly mapped to programs.

3.3.2 Specialization of Needs

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. As a consequence the computer turned from a machine restricted to the purview of the scientists and mathematician into the reach of the personal programmer (*Context*).

As suggested by their names, these first-generation programming languages were primarily aimed for specific scientific and engineering applications and were therefore mainly developed to allow the programmer to write mathematical formulas. When computers became more powerful, the potential needs grow and in parallel the range of applications got broader [Moreau 86]. This resulted in the shift of the kind of abstraction mechanism in programming languages to algorithmic decomposition rather than on mathematical expressions. When the need for data processing applications in business was initiated (*Need*), COBOL (Common Business Oriented Language) was developed in 1960. Technology followed the needs and made attempts to provide satisfactory solutions. In 1965, a language called PL/1 was developed which basically combined the concepts of FORTRAN, ALGOL and COBOL to provide a single general-purpose language suitable for both scientific and commercial purposes [Bergin & Gibson 96]. It is not clear whether this technological step arose from an urgent need, but clearly the language did not follow the expectations in many terms. Nevertheless, it initiated the development of general-purpose languages (*Initiate*).

In parallel with the growing range of complex problems the demand for manipulation of more kinds of data increased (*Need*). Existing languages had already evolved with support for structured data types but this was not sufficient to conveniently express the different kinds of data types required by many applications. Soon the concept of abstract data types and object-oriented programming were introduced (*Solution Domain Knowledge*) and included in languages such as Simula that was intended as a special-purpose language for programming simulations. Abstract data types provided programmers the means to express custom-defined relations between the various kinds of data. Alan Kay at Xerox Parc introduced Smalltalk [Goldberg & Robson 83], a successor of Simula, which was the first language to make full use of object-oriented concepts.

In the early 1990s, Java was developed by Sun as an object-oriented language for programming-in-the-large on multiple platforms [Arnold & Gosling 97]. It became widely known mainly because it provides means to run programs on the internet web browser.

3.3.3 Development of Computer Science Knowledge

Simultaneously with the developments of programming languages, a theoretical basis for these was developed by Noam Chomsky [Chomsky 59][Chomsky 65] and others (*Solution Domain Knowledge*).

Wirth introduced the concept of stepwise refinement [Wirth 71a] of program construction and developed the teaching procedural language Pascal [Wirth 71b] for this purpose. Dijkstra introduced the concept of structured programming [Dijkstra 69]. Parnas addressed the concepts of information hiding and modules [Parnas 72]. Knuth presented a comprehensive overview of a wide variety of algorithms and the analysis of them [Knuth 97].

3.3.4 Emergence of Solution Description Techniques

These publications and the available programming languages that adopted algorithmic abstraction and decomposition have supported the introduction of many structured design methods [Jackson 75][DeMarco 78][Yourdon & Constantine 79] during the 1970s 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 91][Rumbaugh et al. 91][Coad & Yourdon 88] to fit the existing object-oriented language abstractions. CASE tools were introduced in the mid 1980s to provide automated support for structured software development methods [Chikofsky 89][Gane 90]. This had been made economically feasible through the development of graphically oriented computers. Inspired from architecture design [Alexander et al. 77] more recently *design patterns* [Gamma et al. 95] have been introduced as a way to cope with recurring design problems in a systematic way. *Software architectures* [Shaw and Garlan 96] 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 [Szyperki 98][Nierstrasz & Tsichritzis 95]. 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. 98] has been introduced for standardization of object-oriented design models.

4. Project Perspective of Problem Solving

In the previous section we have used the CPC model to analyze the concepts of engineering from an historical perspective. In this section we will analyze problem solving from a project perspective which is the application of the CPC model to current practices. Section 4.1 will focus on mature engineering, section 4.2 will focus on software engineering. This study will enable us to position each engineering discipline improve our understanding about the missing concepts of software engineering. As we described before we consider the mature engineering disciplines as civil engineering, electrical engineering, mechanical engineering and chemical engineering. Our comparison identifies and makes explicit the commonalities and differences between mature engineering and software engineering.

4.1 Project Perspective of Problem Solving in Mature Engineering

4.1.1 Engineering as Problem Solving

In the previous sections we have already noted that mature engineering is problem solving and as such it conforms to the CPC model. To understand how mature engineering implements and specializes the CPC model we have performed a thorough literature study on mature engineering disciplines. We have studied selected handbooks including chemical engineering handbook [Perry et al. 84], mechanical engineering handbook [Marks 87], electrical engineering handbook [Dorf 97] and civil engineering handbook [Chen 95]. Further we have studied several textbooks on the corresponding engineering methodologies of mechanical engineering and civil engineering [Ertas & Jones][Cross 89][Jones 92][Smith et al. 83], electrical engineering [Wilcox et al. 90] and chemical engineering [Biegler 97]. From this study we could detail the CPC model as presented in Figure 2.

We term this model as the *CPC-Engineering model*. Note that this model conforms structurally to the CPC model but in addition defines the engineering specific concepts and functions. The concepts *Synthesis* and *Alternative(s)* are not explicit in the original CPC model but are specific to the engineering model. The concept *Synthesis* represents a refinement of the the functions *Search*, *Apply* and the concept *Solution Description*. In addition the concepts of the control part have been refined for engineering.

Let us now consider these refinements in more detail.

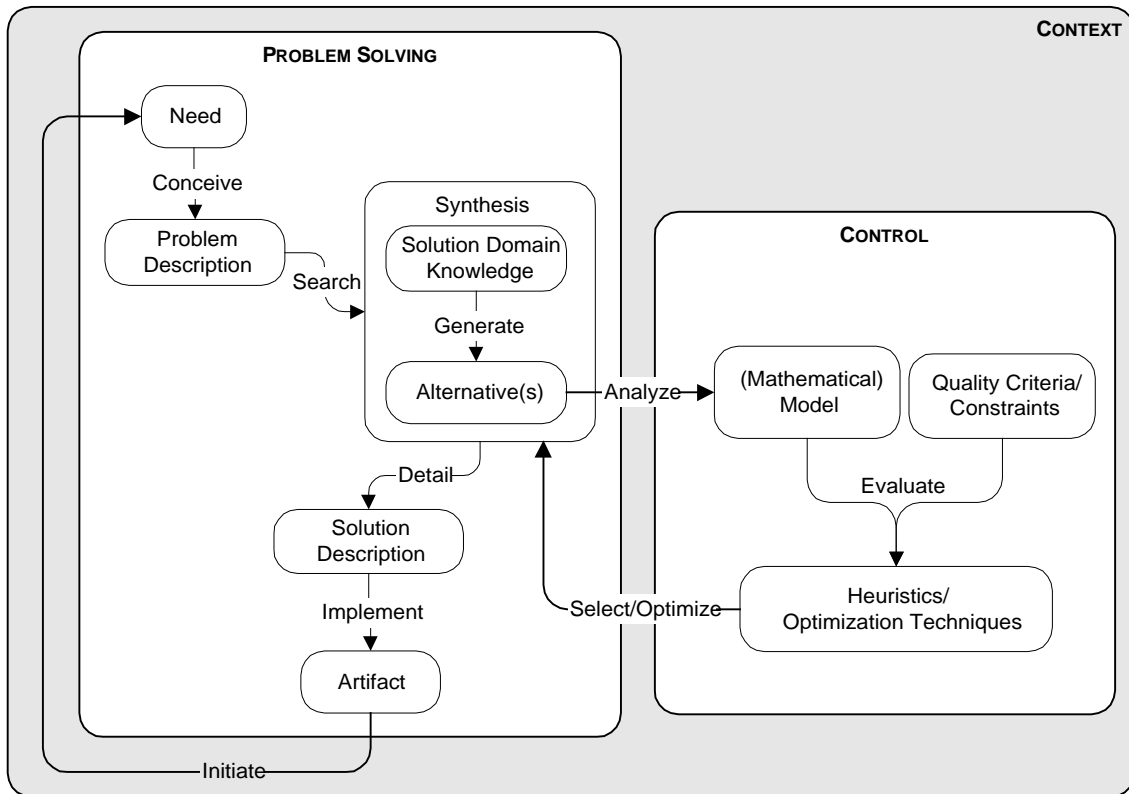


Figure 3. A conceptual model of engineering based on the CPC model: CPC-Engineering model

4.1.2 Conceiving Needs and Describing Problems

Every problem solving process starts with a recognition of the concept *Need*. This is not different for engineering. From our study it follows that a similar set of methods are applied for conceiving the needs in engineering (*Conceive*) and aim to specify the problem as precise as possible. These methods include, for example, interviewing the clients, questionnaires, and investigating related literature. The result of these methods is the representations of the needs in the concept *Problem Description* as it is illustrated in the CPC-engineering model.

Although initial client problems are ill-defined [Rittel & Webber 84] 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 objectives are often ordered into higher and lower-level objectives. The criteria and constraints are often expressed in mathematical formulas and equations. The quality concept is thus explicit in the problem description and refer to the variables and units defined by the International Systems of units (SI). For electrical engineering typical requirements include, for example, *current*, *charge*, *voltage* and *power*. Mechanical engineering and civil engineering problem statements include quantitative requirements like, for example, for *force*, *momentum* and *velocity*. Finally, chemical engineering problem descriptions include requirements for *energy*, *volume*, *pressure* and *temperature*. Some of these variables are used by more than one engineering disciplines other variables are more

specific to a particular engineering discipline. What matters, though, is that problem descriptions include quantified criteria and constraints and that quality is made explicit in this way. From the given specification the engineers can easily calculate the feasibility. In advance, often the different alternatives are defined and even the economical cost of the final product is calculated. If the customer agrees with the calculated costs and the quality of the described product subsequent actions are undertaken.

4.1.3 Synthesis

In this section we will focus on the *Synthesis* process. We will first describe what kind of knowledge (*Solution Domain Knowledge*) is used in the mature engineering disciplines and how this is applied (*Apply*) to solve the problem. Subsequently we will describe the function *Generate* and the concept *Alternative(s)*.

Solution Domain Knowledge

It appears from our study that each mature engineering is based on a rich scientific knowledge that has developed over several centuries. The basic scientific knowledge domain of mechanical engineering and civil engineering is physics which fundamentals are initiated by Newton over 300 years ago. The basic scientific knowledge for chemical engineering is chemistry, which has been formed over the last millennia almost in parallel with chemical engineering. Electrical engineering is merely based on electromagnetic theory defined by Maxwell and others over more than 100 year. These scientific knowledge domains have been improved and specialized since their initial foundations and a wide range consensus on the corresponding concepts has been reached among the experts in the field.

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 more than 2000 pages each and provide a comprehensive coverage in-depth of the various aspects of the corresponding engineering field. In addition, these handbooks are developed with contributions of dozens of top experts in the field indicating a consensus on the fundamental concepts. 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. The handbooks not only describe properties of primitive elements such as material and energy but in addition describe well-known systems at a more gross level such as machines and mechanisms in mechanical engineering, control systems in electrical engineering, bridge design in civil engineering, and process design in chemical engineering. Together with engineering manuals they cover a wide range of scientific, mathematical and technological knowledge. Obviously, scientific knowledge plays an important role in the degree of maturity of the corresponding engineering.

Alternative Generation

Alternative generation is considered as the most important and creative part of the synthesis process. This phase includes decisions on the arrangements of the components, the way how they are linked together, the identification of the component variables, the types of basic dynamic forces, etc. The synthesis problem of the solution can therefore be considered as NP-complete [Maimon & Braha 96][Kalay 87].

The set of alternative design solutions may be very large [Archer 65]. These design alternatives are sometimes directly derived from the related literature, or composed from existing components for which extensive analyses are given in the related literature. In case no accurate formal expressions or off-the-shelf solutions can be found heuristic rules [Coyne et al. 90][Maher 89][Cross 84] [Reitman 64] are used.

4.1.4 Evaluation of Design Alternatives

The *Synthesis* concept represents the creation of one or more configurations of the artifact using scientific knowledge. 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.

It appears that mathematical models are widely used in mature engineering disciplines. The handbooks we studied each contain several chapters on mathematical theories basically on optimization. The selected alternatives are analyzed and evaluated using mathematical techniques such as differential calculus, linear programming, non-linear programming and dynamic programming.

To give an intuition of the synthesis process and the selection of an optimum alternative in mature engineering consider for example a case study in chemical engineering that has been described in [Biegler et al. 97]. The problem is to define a process design for producing ethylene to 190 proof ethanol. To analyze the problem better, they refer to two technical encyclopedias for chemical industry to derive the chemical process reactions to solve the required problem. From literature they then analyze for the chemical elements crucial properties such as the cooking point and the waste produced by the chemical processes. For this purpose they analyze the tables which give accurate values of these properties and derive the exact values for the properties. At this point they can calculate whether there will be a feasible solution within the given cost limits. After the thorough

analysis the synthesis phase is started. Thereby they sketch a flow diagram, which they directly derive also from the literature. After this they analyze the flow diagram and represent mathematical models of the various alternatives by using again the tables, formulas and equations from the literature. Together with the given constraints in the problem description and the expected quality criteria they select the most feasible alternative using the mathematical optimization techniques that have been described in, for example, the chemical handbook [Perry et al. 84].

The other mature engineering disciplines show a similar process as it has been described above.

4.1.5 Solution Description

Engineering disciplines use various kinds of representations to depict the different aspects of the artifact. The form of the solution description is usually represented through textual, graphical or mathematical representations. Mathematical representations are basically used to analyze and evaluate design descriptions. In addition solution descriptions are used to provide different abstractions of the artifact, such as the structural view, dynamic view and functional view [Braha & Maimon 97][Budgen 94][Dym 94].

Each engineering disciplines has its own specific artifact descriptions. In electrical engineering these are for example for electrical circuit diagrams; in mechanical engineering diagrams are used for example, to represent a hydraulic system. In civil engineering descriptions are used to represent, for example the physical architecture of artifacts such as a bridge. In chemical engineering solution descriptions are used, for example, for representing the chemical flow diagrams.

These solution descriptions are used to realize the artifact, which is represented by the function *Implement*. The realization of an artifact is also specific to each engineering discipline and is referred to by various names among which production, manufacturing, realization, building etc.

4.1.6 Decomposition of Problem Solving Process into Phases

In this section we will describe the function *Initiate* of the CPC-Engineering model.

Engineering problems are complex and include many and different kinds of concerns. A problem may include various needs, require different kinds of solution domain knowledge, various goals, different abstractions etc. For large and complex problems it is just practically impossible to cope with all these concerns at a time and by the same engineers. This means that the problem cannot be solved in one step. A traditional technique for coping with complexity is decomposition of the problem into sub-problems. The engineering disciplines apply this technique and decompose the overall engineering process into so-called *phases*. A phase represents a set of related activities to solve a particular problem. As such each phase can itself be modeled using the CPC-engineering model. The decomposition into different phases may be modeled through the function *Initiate*. Each phase results in an intermediate artifact description that will be used to produce subsequent artifact descriptions.

This process will be continued until the final artifact, that is, the artifact directly used by the end-user, is produced. These observations are shown in Figure 4.

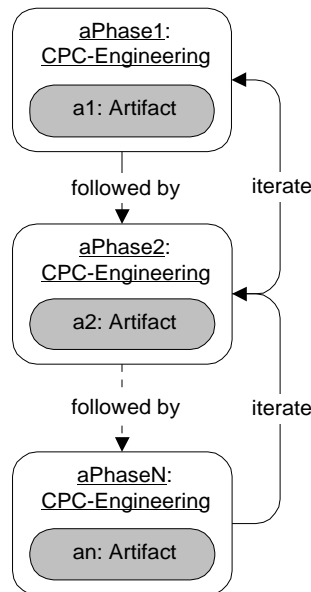


Figure 4. The decomposition of the overall problem solving process into phases

In the figure instances of the CPC-Engineering model are represented through rounded rectangles with underlined names.

Although mature engineering disciplines use different names for the different phases we can observe that in essence they apply a decomposition of the problem solving process into similar phases. From our study we could distinguish, for example, the following common phases: The *Analysis* phase focuses on understanding the problem and the identification of the quality criteria and constraints. The *Conceptual Design* phase focuses on the development of a set of broad solutions based on the problem description. The phase *Detailed Design* intends to develop an artifact description that completely describes the artifact so that it can be realized. The phase *Implementation* realizes the detailed artifact description. Its output is the final end-user artifact. Note that the phases generally correspond to the individual concepts and functions in the CPC model. This shows the scalability potential of the model.

Since each phase is a problem solving process it adopts the concepts and functions as described by the engineering model in Figure 4. The concepts and functions, however, will have different and particular content. For example, the concept *Solution Domain Knowledge* will be used in each phase but the kind of knowledge will be generally different for each phase.

Obviously, the decomposition of the overall process into several phases with a particular concern facilitates the problem solving effort. However, executing the whole process sequentially, that is, phase after phase, is generally complicated and therefore the mature engineering disciplines propose *iteration* between different phases as a necessary step. In Figure 4 iteration is represented by feedback

arrows between the different phases. Iteration enables to check whether the subsequent engineering steps are feasible with the previous engineering decisions and allows to update and/or recover earlier decisions. There are various engineering process models each with its specific iteration flows among the phases. It is out of the scope of the thesis to describe these in detail. What is interesting, though, is the fact that all mature engineering disciplines somehow require iteration for cost-effective problem solving. Note that the iteration in the CPC model may be represented by the function *Initiate*. Before, we noticed that the function *Initiate* may also represent the decomposition of the problem solving process.

4.1.7 Categorizing Engineering Processes

Engineering design processes can be categorized in various ways. A widely used categorization distinguishes *routine design* and *creative design* [Sriram et al. 89][Brown & Chandrasekaran 85][Coyne et al. 87][Dym 94]. Very often, innovative design is considered as a category between routine design and creative design. Although there is agreement on the classification, the criteria for each category is defined in various ways in different engineering disciplines.

However, by abstracting from these studies we consider the following aspects as important criteria for distinguishing whether the design process can be categorized as routine or creative.

- Nature of engineering problems; (*Problem Description*)
- Nature of existing knowledge (*Solution Domain Knowledge*)
- Nature of the methods for analyzing and using knowledge (*Search*)
- Nature of design process (*Apply*)

There are different types of problems, which can be solved in different ways. Problems have been categorized in various ways. A widely adopted categorization distinguishes *well-structured* problems from *ill-structured* problems [Simon 81][Dasgupta 91]. Typical examples of well-structured problems are mathematical problems, such as solving linear equations. Well-structured problems have well-defined problem descriptions, sufficient knowledge for providing cost-effective solutions and criteria to test solutions on their validity. Engineering problems have been usually characterized as ill-structured. Ill-structured problems have the opposite features of well-structured problems. There is no definitive formulation of the problem and the problem statement includes goals and constraints that are vague and/or unknown. It is likely that the problem statement is internally inconsistent and includes conflicts. Further, there is lack of complete knowledge required to solve the problem and/or is not organized for direct use. Suitable criteria are missing for testing when the solution to a problem is found. In addition, there is no definitive solution to the design problem and several equally valid solutions to the same problem may be found.

Note that the concepts *Problem Description* and *Solution Domain Knowledge* form the input for the function *Apply*. Based on these criteria we propose the following meanings of *creative design*, *routine design* and *innovative design*.

Creative design is typically characterized by an ill-structured problem description that includes a vague goal. In addition there is a lack of domain specific knowledge that is needed to generate the set of design solutions. Finally, the design process itself is not well-supported through useful representation forms and heuristic rules and the process is largely based on trial-and-error. Evidently, this is difficult and therefore creative design appears seldom and less frequently. Moreover, usually it is less successful in providing cost-effective solutions. This type of design appears in pre-mature engineering as well as in the fine arts.

Routine design implies a rational design process in contrast to creative design, which relies on conceptual creativity. Routine design is characterized as a design process in which everything that is needed to produce a design is explicit, available and accessible before completing the design. The designer's task is essentially to search for the appropriate alternatives in a well-defined state space of possible designs. The problem is well-structured, all the needed knowledge is available, sufficient methods for reusing knowledge exist and the design process is supported by sufficiently expressive representation forms and useful heuristic rules.

Innovative design lies between creative design and routine design because the problem is not completely well-structured, the required knowledge is lacking, knowledge engineering techniques are weak or the design process is insufficiently supported by useful abstractions and heuristic rules. Hence, still a certain amount of creativity is needed for this kind of design processes.

From the above it follows that creative design is at the one extreme end and routine design is at the other extreme end of the design spectrum. At the creative end of the design spectrum design is best characterized as spontaneous, fuzzy, chaotic and imaginative [Sriram et al. 89]. At the routine end of the design spectrum, the design is predetermined, precise, crisp, systematic and mathematical.

Apparently, each design problem will include all of the three design types. It should be noted that the boundary between creativity and routine design is very difficult to grasp. The fundamental reason behind this is that we cannot understand and model creativity. In addition, the boundary between creativity and routine is a changing one and depends on the designer's experience. The more knowledge is available, the more dedicated designs may be produced that would be considered creative before. In short, the above terms of creative design, innovative design and routine design are relative terms. Nevertheless, it is still useful to adopt such a classification as above when we talk about the maturity degree of engineering. In mature engineering disciplines often problems are more well-structured, knowledge is mature, powerful methods exist for using this knowledge, and the design process includes useful abstractions and heuristic rules that have been formed over a long period of time.

4.2 Project Perspective of Problem Solving in Software Engineering

We will now compare software engineering with the mature engineering disciplines. For software engineering we analyzed a selected set of textbooks [Pressman 94], [Ghezzi et al. 91]. Further, we studied books on popular methodologies [Jacobson et al. 99], [Jones and Shaw 90], [Rumbaugh et al. 91]. We could not find similar handbooks as in mature engineering.

4.2.1 Conceiving Needs

In software engineering, the phase for conceiving the needs is referred to as *requirements analysis*. Very often the basic assumption in software engineering practices is that the customer has already developed a detailed textual requirement specification that entails the basic needs. By means of interviews the requirement specification is then refined and completed.

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. Very often a distinction is made between *functional requirements* and *non-functional requirements*. As described in [Jacobson et al. 99] functional requirements express the actions that a system must perform without considering the constraints. Non-functional requirements impose constraints on functional requirements and specify the required system properties, such as environmental, implementation and performance constraints and the expected quality criteria like maintainability and reliability. In contrast to mature engineering disciplines, however, constraints and the requirements are not expressed in quantified terms. Rather the quality concern is 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. Requirement specifications that would be accepted by neither the customer nor the developer in mature engineering disciplines are very common in software engineering.

4.2.2 Solution Domain Knowledge

Let us now consider the organization and the use of knowledge for software engineering. The field of software engineering is only about 40 years old and obviously has not yet experienced the full maturation of the scientific and technological knowledge as in the traditional engineering disciplines. The basic scientific knowledge on which software engineering relies, is mainly computer science which has developed over the last decades but in many aspects not yet matured. Progress and maturation have been made only in isolated parts, such as algorithms and abstract data types [Shaw & Garlan 96]. Knowledge on algorithms, such as sorting and searching, have been compiled in the book of Knuth [Knuth 97] and are widely applied. Similarly several theories and principles, such as modularity and information hiding, have been published on the notion of abstract data types.

Compiler construction is one of the basic applications of software engineering that has reached practically a stable state and much of its knowledge is reused.

If we relate the quantity of knowledge to the supporting knowledge of mature engineering disciplines, the available knowledge in software engineering which is currently organized and actually used is quite meager. There are, for example, no standard handbooks or manuals like in other 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. Let us now consider how problems are actually solved in software engineering.

It turns out that a common implicit assumption of the current approaches in software development is that the concept *Problem Description*, or requirement specifications, form the basic input for the development of software solutions and scientific knowledge has only a minor role. The general idea is that requirements have to be specified using some representation [Webster 88] and this should be refined along the software development process until the final software is delivered. Software development is thus seen as an evolutionary transformation process of the initial requirements until the final software. After the NATO conference in 1968 this remained more or less the general attitude in software development and this trend continues until recently.

"Software design translates the requirements for the software into a set of representations that describe data structure, architecture, algorithmic procedure, and interface characteristics" [Pressman 92]

"A software development process is the set of activities needed to transform a user's requirements into a software system" [Jacobson et al. 99]

"... through a series of reification (refinement) steps, the specification is transformed into an implementation which satisfies the specification." [Jones & Shaw 90]

Consider for example the wide range of software development practices that adopt objects. Objects are abstract data types and have been introduced together with the corresponding software methods to express the requirements specification to the solutions in an organized way. The approach for deriving a solution remained the same, however. That is, by expressing requirements in some specification and then after a number of iteration steps transform these to a set of objects.

This approach whereby the concept of *Solution Domain Knowledge* is not directly explicit, is certainly different than the approach from mature engineering disciplines whereby scientific knowledge plays a fundamental role. This approach resembles the early pre-mature phases of traditional engineering disciplines when scientific knowledge was not mature yet or not applied in practice.

Although, this view of software development may have been suitable for well-defined numerical calculation problems of the early days, we need to question whether it is still valid for the current and future applications. Since then software development problems have got larger and more complex.

From practice it follows that some engineers differ in the background and experience they have and as such provide artifacts with different qualities. Experimental and empirical studies [Curtis et al. 88][Adelson & Soloway 85] have shown that one of the fundamental aspects of exceptional engineers is that they are familiar with the application domain, which enables them to map between problem structures and solution structures easily. Obviously, knowledge in software engineering has the same important role as in other engineering disciplines but unfortunately this does not reflect to current practices. There is, however, an increasing consciousness in the software engineering community about the role of knowledge, as it is apparent in the following:

“In current software practice, knowledge about techniques that work is not shared effectively with workers on late projects, nor is there a large body of software development knowledge organized for ready reference. Computer science has contributed some relevant theory, but practice proceeds largely independently of this organized knowledge” [Shaw & Garlan 96]

“Before the program can be written, humans have to describe and organize the knowledge it represents according to specific knowledge sources [Robillard 99]

Recently, *domain analysis* is introduced as the process of identifying, capturing and organizing domain knowledge about the problem domain with the purpose of making it reusable when creating new systems [Prieto-Diaz 91][Arrango 94].

4.2.3 Alternative Generation

Software engineering problems can also be classified in the category of NP-complete problems [Garey & Johnson 79]. The concept of *Alternative(s)*, however, is not explicit in software engineering. In the previous section we have seen that unlike mature engineering disciplines software engineering is not supported by an extensive amount of knowledge and usually the process for deriving solutions is basically based on the problem description. To support this process heuristic rules and design patterns are applied. A number of methods with a large set of heuristic rules exist, though, there is no common agreement on the kind of heuristic rules which show enough potential to be standardized in a common software engineering method. The goal of design 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.

The selection and evaluation of design alternatives in mature engineering disciplines is based on quantitative analysis through optimization theory of mathematics. Apparently, 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 basically an informal basis. 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 & Phleeger 97]. There is however a broad agreement that quality should be taken into account when deriving solutions. In software engineering quality factors are often divided into *external* and *internal* qualities corresponding the distinction between internal and external attributes of entities. The external qualities are visible to the end-users of the system. The internal qualities concern the developers of the software system. Internal qualities deal largely with the structure of the system and help to achieve the external qualities. Quality factors may be attributed to the process, the product and the available resources [Fenton & Phleeger 97]. Several quality factors are described in the software engineering literature. Some important software quality factors such correctness, robustness, reliability, adaptability, reusability and extensibility are defined [Humphrey 89][Ghezzi et al. 91]. However, the bottom line is that these quality factors are not quantified and as such cannot be explicitly used to generate, evaluate and optimize design alternatives, which is a common practice in mature engineering disciplines.

5. 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 [Walker 96][Pressman 94][Brooks87][Booch 91]. 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.

In this paper we adopted a broader view and argued that software engineering is in essence an engineering discipline and engineering on its turn is fundamentally a problem solving process. Problem solving has been extensively studied in cognitive sciences such as [Smith & Browne 93][Agre 82][Rubinstein & Pfeiffer 80][Newell & Simon 76]. These studies consider problem solving not as a random process based on trial-and-error but rather as a process that involves a series of recurring mental processes. The work of Newell and Simon has long dominated the theories on problem solving in cognitive science, which they have also applied to reflect on engineering design. They describe the engineering design process as a problem-solving process of searching through a state space in which the states represent the design solutions. Thereby, the goal of the problem is analogous to the final state in the maze that represents the solution of the problem. The search through this state space involves making decisions based on the goals and constraints that exclude some infeasible solutions. Newell and Simon described an automated problem solving system called

General Problem Solver (GPS) that develops a computer program for the solution of well-defined problems. These studies are basically concerned with problem solving from a psychological perspective and attempt to understand the human thinking processes. Moreover, they do not explicitly consider the control aspects in problem solving. The CPC model presented in this paper uniquely provides a conceptual model that integrates control and problem solving in a context. It is not a cognitive model but rather it defines the separate product concepts involved in problem solving and control. This makes it more suitable to express and reason about the engineering problem solving process.

We have applied the CPC model for describing problem solving from a historical perspective. Several publications consider the history of computer science [Nijholt & Ende 94][Moreau 86][IEEE AnnalsHC] providing a useful factual overview of the main events in the history of computer science and software engineering. The paper from Shapiro [Shapiro 97] 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 required¹⁵.

A cross-disciplinary seminar on the history of software engineering has been organized in 1996 in Dagstuhl, Germany, [Brennecke & Keil-Slawik 96] where about a dozen of historians met with about a dozen computer scientists to discuss the history of software engineering. The report of the seminar, though, does not provide any concise historical analysis or results that lead to a deeper understanding of the essence of software engineering and the software crisis.

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 maintains in his book [Petroski 92] that lessons learned from failures can substantially advance engineering. In alignment with this, the paper [Holloway 99] derives practical hints for software development from two well-know failures in traditional engineering disciplines - the collapse of The Tacoma Narrows Bridge in 1940 and the destruction of the space shuttle Challenger in 1986. In [Baber 90] Baber 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 who 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

¹⁵ This kind of motivation is similar as to the philosophical thought of eclecticism that argues for adopt multiple thoughts rather than a single thought to be successful.

whatever such basis may exist. Shaw 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.” [Shaw 90]

Using her model she compares civil engineering and chemical engineering and concludes that these engineering disciplines have matured basically because of the supporting science that has evolved. Shaw basically distinguishes between craft, commercial and professional engineering processes. These distinct engineering states can be each expressed as a different instantiation of the CPC model. The immature craft engineering process will lack some of the concepts as described by the CPC model. The mature professional engineering process will include all the concepts of the CPC model.

In [Fenton et al. 94] the authors criticize the lack of well-designed experiments for measurement-based assessment in software engineering. They state that currently the evaluation of software engineering practices basically 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 CPC model 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.

The CPC model has been used as a reference model for our comparative analysis of philosophy and engineering. There have been incidental comparative studies as from the ones described above that compare mature engineering disciplines with software engineering disciplines. To the best of our knowledge there do not exist studies that discuss and compare philosophy with engineering.

6. Conclusion

The thesis of this paper is that the fundamental problems the current software engineering discipline has to cope with are rather conceptual than technical. To provide a thorough understanding of the missing concepts we maintained that it is necessary to have a broader perspective beyond the software engineering discipline. 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 philosophy, mature engineering and software engineering. In addition we have presented an analysis of the mature engineering disciplines from a contemporary project perspective. This has enabled us to position the software engineering discipline and validate its maturity level. These thorough conceptual and comparative analyses have resulted in the following 14 conclusions:

1. Mature problem solving conforms to the CPC model

Engineering and philosophy both conform to the CPC model. In the sections on the historical perspectives of problem solving in history and mature engineering it appears that the fundamental concepts and functions of problem solving were initially not explicit but have emerged over time. We observed that in the early historical phases problem solving was primitive and can be explained by almost a direct match from the concept *Need* to the concept *Artifact*. In philosophy, we have seen that before the first philosophers, people mainly adopted mythological explanations for the different phenomena's in life. Philosophy as such started with a break with these mythological explanations and the consciousness that problems had to be solved through rational thinking instead. In engineering, we observed a similar pattern. The initial phases of engineering can be characterized as craft work in which producing artifacts had no any systematic or scientific basis but was mainly based on speculative thinking, practical know-how, intuition and trial-and-error. Hence, we can state that problem solving was an unconscious process at this stage as well. When the consciousness about the problem solving process increased we can see that the concepts of the CPC model started to emerge as an explicit concern and matured over time in both philosophy and engineering.

2. Artificial solutions cannot be enforced but must be derived from the basic needs

Obviously, *Need* is the basic concept of the CPC model that plays a major role and as such is the motivator in philosophy and engineering. From our studies it appears that the need concept is generally an unsatisfactory and abstract state. For eliminating the undesirable situation and satisfying needs, it is first required that the problem is externalized in some description. Thereby, problems need to be stated by defining the need in terms of the state description, so that the current state of unsatisfactory affairs is changed to a satisfactory state. This is to say that the concept *Problem Description* needs to be an explicit concern. An important issue thereby is that the problem description should reflect the real needs. Problem descriptions that are based on artificial or non-urgent needs, will lead to solutions that are dispensable. In philosophy we observe that the writings of the philosophers directly match the need in their contemporary context and attempted to find answers for the life questions initiated by this context. The early Greek, for example, who were basically interested in astronomy and nature attempted to find the basic element of the universe. Later when social problems arose, philosophers turned their attention to morality and ethic issues. In the same way, the subsequent periods of philosophy show the direct alignment of philosophical writings on the needs that were initiated by the context. In engineering, we can observe a similar pattern in which useful problem solving relies on addressing the right needs. In an agricultural society, for example, the basic needs like housing and food supply were necessary and as such engineering had to address these needs by producing shelter, tools and weapons. Later with the rise of the cities communication and trade became a necessary need and accordingly engineers produced roads, canals and bridges. History teaches that artificial needs, that is, needs that are not linked to actual existing needs, will

mostly fail. An example of a less urgent need in software engineering is, for example, the development of the programming language PL/1 that combined the concepts of existing languages but lacked the expected popularity due to the less urgent needs that it addressed.

3. The specialization of needs has led to the specialization of the disciplines

We have seen that the context continuously changes and impacted the evolution of philosophy, mature engineering and software engineering. Following the changes in the context, for example due to scientific or social developments, the need has changed over the period and resulted in specializations of the disciplines. In philosophy, the specialization of needs has resulted in the emergence of different philosophical branches and thought systems, such as the early Greek sophism, epicureanism, skepticism and stoicism and the medieval rationalism, empiricism and positivism. In engineering, the specialization of needs has basically resulted in the specialization of engineering disciplines such as mechanical engineering, civil engineering, chemical engineering and electrical engineering. Specializations of needs occur also within one engineering discipline. The engineering disciplines we considered decompose the problem solving process into several phases that each address a different goal or needs.

4. Solution description is necessary to cope with complexity

With the increasing complexity of the problems, individuals fail to be effective in problem solving due to the psychological limitations of the storage and processing speed of the human mind. For this reason, it became necessary to make the concept *Solution Description* explicit. In engineering, this concept refers to descriptions of the artifact. This concept enabled to communicate about the artifact before production, evaluate it and use it as guidance for production. In philosophy the distinction between the solution description and the artifact is not clear. We may consider the writings of the philosophers as the solution description and the application to it on practical life as its implementation.

5. Preserving and communication of solution domain knowledge is essential for mature problem solving

In the history of philosophy and engineering we have observed that when the focus on knowledge is lost, problem solving is based on more primitive techniques and fails to be successful. The preservation, codification and communication of knowledge are fundamental to effective problem solving. Once *Solution Domain Knowledge* concept becomes explicit it can have huge positive effects. In philosophy, for instance, the Renaissance movement was a result of a renewed interest in the classical thought and science which had been codified, further developed and communicated to the West by the scientists and philosopher from the East. In the history of engineering knowledge has played a decisive role. Although several inventions, such as the steam engine, were invented through a series of trial-and-error experiments, it was the accumulation and communication that improved the systematic approach to producing cost-effective artifacts. For example, the basis for contemporary

civil engineering and mechanical engineering are derived from the physical laws of Newton and others.

6. Maturity of the control concepts indicates the maturity of the overall problem solving process

A similar development of the maturation of the problem solving concepts we may observe in the control concepts. Control is directly related to an explicit understanding and interpretation of the problem solving process. After the people got conscious about the problem solving concepts and explicitly reasoned about these, the next level consciousness started to develop with the control concepts. Philosophy, initially started with the break with mythology and focused on rational explanations on the nature of things that formed the problem solving process in philosophical thought. Philosophical studies on interpretation further developed over time and after a while philosophers also focused on the understanding of interpretation, that is control of problem solving. This philosophical thought is expressed in the hermeneutic philosophy, which promotes that for proper understanding the context of the artifact needs to be grasped. This is to say that hermeneutics actually conforms to CPC model, which is based on the assumption that a controlled problem solving is valid within a context. Engineering has showed the same maturation process. After the separate problem solving concepts were made explicit, the subsequent focus was on the control concepts. This increased consciousness on control is manifested with the emergence of the computer-aided design (CAD) and computer-aided manufacturing techniques (CAM).

7. History is a continuous problem solving process

From our study on philosophy and engineering, it appears that history is a controlled problem solving process operating in a specific context. Although the context is difficult to express our historical analysis shows that both of philosophy and engineering are continuously attempting to solve the important problems that arise out of the contemporary context. Extrapolating these results we can assume that the future will show the same pattern. The CPC model distinguishes and makes explicit the problem solving concepts and likewise enables to describe, interpret or even predict the state of the affairs of a given discipline. In this paper we focused on software engineering and could represent a clear view of its maturity level.

8. Interplay between engineering and philosophy has matured problem solving; useful concepts may be derived from philosophy to improve engineering.

Problem solving matured through a continuous interpretation and improvement within the disciplines philosophy and engineering. In addition, a certain interplay across the disciplines has influenced and enriched the common problem solving approach. In philosophy, many philosophers have contributed to the systematization and accumulation of knowledge, that is, the concept *Solution Domain Knowledge* became explicit in problem solving. In addition, the techniques for knowledge accumulation and logical justifications (*Search*) have been improved and the notions of *concept*,

paradigm, abstraction, generalization, and specialization have been formed. This has significantly contributed to the maturation of the engineering disciplines. History has proven that it might thus be worthwhile to consider concepts in philosophy to identify the deficiencies of current engineering and likewise improve it. On the other hand, developments in engineering have had a great impact on philosophical movements. Some philosophies even became obsolete, after improvements in engineering. For example, the classic Greek natural philosophy became obsolete since the 17th century when new mathematically and empirically underpinned concept of motion was defined. More recently, in this age, computer science and software engineering is dramatically changing the world by providing automatic support for research and education activities. Advances in computing are having also a significant impact upon foundational concepts in philosophy, such as the mind, consciousness, reasoning, logic, knowledge, truth and creativity [Bynum & Moor 99].

9. *Mature engineering is supported by extensive scientific knowledge and is therefore more routine than software engineering which has a meager amount of scientific knowledge base*

Knowledge is the formalization of past experiences. The more knowledge is available the better the engineer can be directed in producing cost-effective artifacts and the more routine the engineering process. Without knowledge the artifact production process is generally based on intuition and practical know-how. Mature engineering disciplines have collected and formalized a broad range of engineering knowledge in the corresponding handbooks, manuals and encyclopedia. Due to the lack of accumulated and formalized reusable knowledge software engineering practices must more rely on the creativity and practical know how of the software engineering. Obviously, current software engineering practices are, with respect to mature engineering disciplines, more creative than routine.

10. *Mature engineering derives abstractions from solution domain knowledge - Software engineering derives abstractions basically from client requirements*

To cope with the complexity different engineering disciplines propose various approaches. We have seen that scientific knowledge and mathematical knowledge is widely applied in mature engineering disciplines whereby the engineers are guided through well-defined handbooks and manuals. Current software engineering practices adopt a different view and state that software solutions must be found through transforming the requirements specification in several steps. The solution domain knowledge has only a marginal role in software engineering. The reason for this has been explained from the fact that software engineering is a premature engineering discipline with lack of consensus on several fundamental concepts. The scientific knowledge on which software engineering is based has only a short history and as such is not mature yet.

11. *Synthesis in engineering disciplines is NP-complete and heuristics are inherently necessary*

It appeared that the synthesis in engineering processes is an NP-complete problem and this is less a matter of the kind of engineering but rather an inherent aspect of most engineering problems. This

means that engineering problems are not solvable within the given time and resources. For this reason, the use of heuristic rules is necessary to minimize the extremely large solution space. In software engineering, the same situation can be observed. Today, software is written for a wide range of applications, other than numerical calculations and indeed software has grown in size and complexity.

*12. Mathematical modeling is used to reduce the solution boundaries in mature engineering -
Solution boundaries in software engineering are not explicit.*

Despite of the NP-completeness of the engineering problems, in mature engineering mathematical modeling techniques, such as calculus, linear programming and dynamic programming are used to reduce the boundaries within which the possible solutions should be considered. Although, not the complete set of alternatives is described at once, it is possible to derive all the possible alternatives. In addition, alternatives that remain outside the solution boundary can be easily omitted. The knowledge on what is possible and what is not possible in advance eases the search for a solution. In software engineering the solution boundaries for a given problem are harder to define. Basically heuristic rules are used to reduce the solution boundaries but mathematical techniques for this purpose are missing. Consequently, the boundaries remain implicit and the search for an adequate solution is therefore more cumbersome than in mature engineering.

*13. Mature engineering evaluates alternatives using rigid mathematical approaches -
Evaluation of alternatives in software engineering is informal and often implicit.*

In addition to techniques for defining solution boundaries, mature engineering disciplines are supported with a set of mathematical optimization techniques to evaluate individual alternatives. In software engineering, the evaluation is largely based on heuristics or implicit criteria and suitable optimization techniques are missing.

*14. Quality concept in mature engineering is explicit and mathematically defined -
Quality concept in software engineering is generally implicit and lacks measurements*

Quality is an explicit concept in mature engineering disciplines and is represented within the different engineering phases. For example, requirement specification in mature engineering includes explicit variables indicating the requiring quality. In software engineering requirement specifications quality is indicated basically through informal statements.

References

- [Adelson & Soloway 85] Adelson, B, & Soloway E. The role of domain experience in software design. *IEEE Trans. Software Engineering*, SE-11(11), 1351-9, 1985
- [Agre 82] Agre, G.P. *The concept of problem*, Educational Studies, vol. 13, pp. 121-142, 1982.

- [Alexander 64] Alexander, C. *Notes on the Synthesis of Form*, Harvard University Press, Cambridge, MA, 1964.
- [Alexander et al. 77] Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., & Angel, S. *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, New York, 1977.
- [Al-Hassan & Hill 86] Al-Hassan, A.Y., & Hill, D.R., *Islamic Technology: an Illustrated History*, Cambridge University Press, 1986.
- [Archer 65] Archer, L.B., *Systematic Method for Designers*, The Design Council, London, also reprinted in [Cross 84], 1965.
- [Arnold & Gosling 97] Arnold, K., & Gosling, J. *The Java Programming Language*, 2nd edition, Addison-Wesley, 1997.
- [Arrango 94] Arrango, G. *Domain Analysis Methods*. in: *Software Engineering Reusability*, R. Schafer, Prieto-Diaz, & M. Matsumoto (Eds.), Ellis Horwood, New York, 1994.
- [Bergin & Gibson 96] Bergin, T.J., & Gibson, R.G (eds.). *History of Programming Languages*, Addison-Wesley, 1996.
- [Biegler et al. 97] Biegler, L.T., Grossmann, I.E., & Westerberg, A.W. *Systematic Methods of Chemical Process Design*, Prentice Hall, 1997.
- [Booch 91] Booch, G. *Object-Oriented Analysis and Design, with Applications*, Redwood City, CA: The Benjamin/Cummins Publishing Company, 1991.
- [Braha & Maimon 97] Braha, D., & Maimon, O. *The Design Process: Properties, Paradigms, and Structure*. IEEE Transactions on Systems, Man, and Cybernetics, Vol. 27, No. 2, March 1997.
- [Brennecke & Keil-Slawik 96] Brennecke, A., & Keil-Slawik, R. *History of Software Engineering*, Dagstuhl seminar, Germany, organized by: Aspray, W., Keil-Slawik, R., & Parnas, D.L. 1996.
- [Brooks 87] Brooks, F.P. Jr. *No silver bullet: Essence and accidents of software engineering*. *IEEE Computer*, 10-19, 1987.
- [Brown & Chandrasekaran 89] Brown, D.C., & Chandrasekaran B. *Design Problem Solving*. London: Pitman, 1989.
- [Budgen 94] Budgen, D. *Software design*, Addison-Wesley, 1994
- [Burstall 63] Burstall, A.F. *A history of Mechanical Engineering*, Faber and Faber, London, 1963.
- [Bynum & Moor 99] Bynum, T.W., & Moor, J (eds.). *The Digital Phoenix: How Computers are Changing Philosophy*, Blackwell Publishers, 1999.
- [Chen 95] Chen, W.F. *The Civil Engineering Handbook*, CRC Press, 1998.
- [Chikofsky 89] Chikofsky, E.J., *Computer-Aided Software Engineering (CASE)*. Washington, D.C. IEEE Computer Society, 1989.
- [Chomsky 59] Chomsky, N., *On certain formal properties of grammars*, *Information and Control* 2,2(1959), 137-167, 1959.
- [Chomsky 65] Chomsky, N. *Aspects of the Theory of Syntax*. MIT Press, 1965.
- [Coad & Yourdon 91] Coad, P., & Yourdon, E. *Object-Oriented Design*, Yourdon Press, 1991.
- [Codd 70] E. F. Codd, *A Relational Model of Data for Large Shared Data Banks*, *Communications of the ACM*, Vol. 13, No. 6, June 1970, pp. 377-387.
- [Coyne et al. 87] Coyne, R.D., Rosenman, M.A., Radford, A.D., & Gero, J.S. *Innovation and Creativity in Knowledge-based CAD*, in: *Expert Systems in Computer-Aided Design*, ed. J.S. Gero, pp. 435-465, North-Holland, Amsterdam, 1987.
- [Coyne et al. 90] Coyne, R.D., Rosenman, M.A., Radford, A.D., Balachandran, M., & Gero, J.S. *Knowledge-based design systems*, Addison-Wesley, 1990.

- [Cross 89] Cross, N. *Engineering Design Methods*, Wiley & Sons, 1989.
- [Cross 84] Cross, N. *Developments in Design Methodology*, Wiley & Sons, 1984.
- [Curtis et al. 88] Curtis, B., Krasner, H., & Iscoe, N. *A field study of the software design process for large systems*. Communications of the ACM, Vol. 31(11), pp. 1268-87, 1988.
- [Dasgupta 91] Dasgupta, S. *Design Theory and Computer Science*. Cambridge University Press, 1991.
- [Date 77] Date, C., *An introduction to Database Systems*, Addison-Wesley, 1977.
- [DeMarco 78] DeMarco, T., *Structured Analysis and System Specification*, Yourdon Inc., 1978.
- [Dijkstra 68] Dijkstra, E. W. *The structure of "THE"-multiprogramming system*. Comm. ACM 11, 5 (May 1968), 341-346.
- [Dijkstra, 69] Dijkstra, E. W., "Structured Programming," *Software Engineering Techniques*, Buxton, J. N., and Randell, B., eds. Brussels, Belgium, NATO Science Committee, 1969.
- [Dorf 97] Dorf, R.C. *The Electrical Engineering Handbook*, New York, Springer Verlag, 1997.
- [Dunsheath 62] Dunsheath, P. *A History of Electrical Engineering*, Faber & Faber, London, 1962.
- [Dym 94] Dym, C. L. *Engineering Design: A synthesis of Views*. Cambridge University Press. 1994.
- [Ertas & Jones 96] Ertas, A., & Jones, J.C. *The Engineering Design Process*, Wiley & sons, 1996.
- [Fenton & Phleeger 97] Fenton, N.E., & Phleeger, S.L. *Software Metrics: A Rigorous & Practical Approach*. PWS Publishing Company, 1997.
- [Fenton et al. 94] Fenton, N.E., Phleeger, S.L. & Glass, R.L. *Science and Substance: A Challenge to Software Engineers*, IEEE Software, pp. 86-95, July 1994.
- [Foerster 79] Foerster, H. Von., *Cybernetics of Cybernetics*, in: Klaus Krippendorff (ed.), *Communication and Control in Society*, New York: Gordon and Breach, 1979.
- [Forbes 58] Forbes, R.J. *Man The Maker: A History of Technology and Engineering*, Constable and Company, London, 1958.
- [Gamma et al. 95] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA, Addison-Wesley, 1995.
- [Gane 90] Gane, C. *Computer-Aided Software Engineering: The Methodologies, the Products, and the Future*, Englewood Cliffs, NJ: Prentice Hall, 1990.
- [Garey & Johnson 79] Garey, M.R., & Johnson, D.S. *Computers and Intractability: A guide to the theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.
- [Garrison 91] Garrison, E. *A history of engineering and technology: Artful Methods*. CRC Press, 1991.
- [Ghezzi et al. 91] Ghezzi, C., Jazayeri, M., & Mandrioli, D. *Fundamentals of Software Engineering*. Prentice-Hall, 1991.
- [Gibbs 94] Gibbs, W.W., *Software's Chronic Crisis*. The Scientific American, pp. 86-95, September, 1994.
- [Goel & Pirolli 89] Goel, V., & Pirolli, P. *Motivating the notion of generic design within information-processing theory: The design problem space*, AI-Magazine, vol. 10, pp. 18-36, 1989.
- [Goldberg & Robson 83] Goldberg, A., & Robson, D. *Smalltalk 80: The Language and its Implementation*, Addison-Wesley, 1983.
- [Hull 59] Hull, L.W.H. *History and Philosophy of Science: An Introduction*. London: Longmans, 1959.
- [Humphrey 89] Humphrey, W.S. *Managing the Software Process*. Oxford: Addison-Wesley, 1989.
- [Hunt 94] Hunt, E., *Problem Solving*, in: *Thinking and Problem Solving*, ed. R.J. Sternberg, pp. 215-232, Academic Press, 1994.
- [IEEE AnnalsHC] *IEEE Annals of the History of Computing journal*.

- [Jacobson et al. 99] Jacobson, I., Booch, G., & Rumbaugh, J. *The Unified Software Development Process*, Addison-Wesley, 1999.
- [Jackson 75] Jackson, M.J., *Principles of Program Design*, Academic Press, 1975.
- [Jones 92] Jones, J.C., *Design Methods: Seeds of human futures*. London: Wiley International, 1992.
- [Jones & Shaw 90] Jones, C.B., & Shaw, R.C., *Case Studies in Systematic Software Development*, Prentice Hall, 1990.
- [Kalay 87] Kalay, Y.E. *Computability of Design*, Y.E. Kalay (Ed.), John Wiley and Sons, New York, 1987.
- [Knuth 97] Knuth, D.E. *The Art of Computer Programming*, Addison-Wesley, 1997.
- [Kolenda 74] Kolenda, K., *Philosophy's journey: a historical introduction*, Reading, Mass : Addison-Wesley, 1974.
- [Krick 69] Krick, E.V. *An introduction to engineering and engineering design*. Wiley & sons, 1969.
- [Marks 87] Marks, L.S., *Mark's Standard Handbook for Mechanical Engineers*. McGraw-Hill, 1987.
- [Maher 89] Maher, M.L. *Synthesis and evaluation of preliminary designs*, in: Artificial Intelligence in Design, J.S. Gero, Ed. New York: Springer-Verlag, 1989.
- Maimon, O., & Braha, D. *On the Complexity of the Design Synthesis Problem*. IEEE Transactions on Systems, Man, and Cybernetics, Vol. 26, No. 1, Jan 1996.
- [MSEncarta 96] *Microsoft Encarta 96 Encyclopedia*, CD-ROM, Microsoft Corporation, 1995.
- [Melchert 95] Melchert, N. *The great conversation: a historical introduction to philosophy*, Mountain View, Mayfield Publ., 1995.
- [Miller 56] Miller, G. *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information*. The Psychological Review. Vol. 63(2), 1956.
- [Moreau 86] Moreau, R. *The Computer Comes of Age: The People, the Hardware, and the Software*, MIT Press, 1986.
- [Naur & Randell 69] Naur, P., & Randell, B. (eds.) *Software engineering: A report on a Conference sponsored by the NATO Science Committee*, NATO, 1969.
- [Neumann 95] Neumann, P.G. *Computer Related Risks*. New York: ACM Press, 1995.
- [Newell & Simon 76] Newell, A., & Simon, H.A., *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, NJ, 1976.
- [Nierstrasz & Tsichritzis 95] Nierstrasz, O., & Tsichritzis, D. *Object-Oriented Software Composition*, Prentice-Hall, 1995.
- [Nijholt & Ende 94] Nijholt, A., & Ende, van den J. *Geschiedenis van de rekenkunst: van kerfstok tot computer*, Academic Service, Schoonhoven, 1994.
- [Parnas 72] Parnas D. L., *On the Criteria to be Used in Decomposing Systems into Modules*, Communications of the ACM, 15 (12), 1972.
- [Palfreman & Swade 93] Palfreman, J., & Swade, D. *The Dream Machine: Exploring the Computer Age*, BBC publications, 1993.
- [Partington 70] Partington, J.R., *A history of chemistry*, London: MacMillan, 1970.
- [Perry et al. 84] Perry, R.H. *Perry's Chemical Engineer's Handbook*, McGraw-Hill, New York, 1984.
- [Petroski 92] Petroski, H. *To Engineer is Human: The Role of Failure in Successful Design*. New York: Vintage Books, 1992.
- [Pressman 94] Pressman, R.S. *Software Engineering: A practitioner's approach*, Mc-Graw-Hill, 1994.
- [Prieto-Diaz & Arrango 91] Prieto-Diaz, R., & Arrango, G. *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, Los Alamitos, CA, 1991.

- [Reitman 64] Reitman, W.R. *Heuristic decision procedures, open constraints, and the structure of ill-defined problems*, in: Human Judgments and Optimality, M.W. Shelly & G.L. Bryan, Eds. New York: Wiley, 1964.
- [Rittel & Webber 84] Rittel, H.W., & Webber, M.M. *Planning problems are wicked problems*, Policy Sciences, 4, 155-169, also reprinted in [Cross 84], 1984.
- [Robillard 99]. Robillard, P.N., *The role of Knowledge in Software Development*, Communications of the ACM, Vol 42, No.1, pp. 87-92, January 1999.
- [Rubinstein & Pfeiffer 80] Rubinstein, M.F. & Pfeiffer, K. *Concepts in Problem solving*. Prentice-Hall, Englewood Cliffs, 1980.
- [Rumbaugh et al. 98] Rumbaugh, J., Jacobson, I., & Booch, G. *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1998.
- [Rumbaugh 91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorenzen, W. *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
- [Shapiro 97] Shapiro, S. *Splitting the Difference: The Historical Necessity of Synthesis in Software Engineering*. IEEE Annals of the History of Computing 19, no. 1, pp. 20-54, 1997.
- [Shaw 90] Shaw, M. *Prospects for an Engineering Discipline of Software*, IEEE Software, pp. 15-24, 1990.
- [Shaw & Garlan 96] Shaw, M., & Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
- [Silberschatz et al. 92] Silberschatz, A., Peterson, J., & Galvin, P. *Operating System Concepts*, Addison-Wesley, 1992.
- [Simon 62] Simon, H.A. *The Architecture of Complexity*. *Proceedings of the American Philosophical Society*. Vol. 106, 1962.
- [Simon 81] Simon, H.A. *The Sciences of the Artificial*, 2nd Edition, MIT Press, Cambridge, MA, 1981.
- [Smith & Browne 93] Smith, G.F., & Browne, G.J. *Conceptual Foundations of Design Problem Solving*. IEEE Transactions on Systems, Man, and Cybernetics, Vol. 23, No. 5, Sept/Oct 1993.
- [Smith et al. 83] Smith, A.A., Hinton, E., & Lewis, R.W., *Civil Engineering Systems Analysis and Design*, Wiley & Sons, 1983.
- [Sriram et al. 89] Sriram, D., Stephanopoulos, G., Logcher, R., Gossard, D., Groleau, N., Serrano, D., and Navinchandra, *Knowledge-based system applications in engineering design: Research at MIT*, AI Magazine, vol. 10, no. 3, pp. 79-96, 1989.
- [Stroustrup 86] Stroustrup, B. *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.
- [Szyperski 98] Szyperski, C. *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998.
- [Umplebey 90] Umplebey, S.A., *The Science of Cybernetics and the Cybernetics of Science*, Cybernetics and Systems, Vol. 21, No. 1, 1990, pp. 109-121, 1990.
- [Upton 75] Upton, N., *An illustrated history of civil engineering*, London : Heinemann, 1975.
- [Visser & Hoc 90] Visser, W., & Hoc, J.M. *Expert software design strategies*. In: Psychology of Programming, Hoc, J.M., Green, T.R.G., Samurçay, R., & Gilmore, D.J. (eds). London: Academic Press.
- [Walker 96] Walker, E. *Software/Hardware Reliability - Bridging the Communication Gap*, RAC Journal, Vol. 4, no. 2, 1996.
- [Webster 88] Webster, D.E. *Mapping the design information representation terrain*. *IEEE Computer*, 21(12), 8-23, 1988.
- [Websters] Webster's Dictionary.

- [Wilcox et al. 90] Wilcox, A.D, Huelsman, L.P, Marshall, S.V., Philips, C.L., Rashid, M.H., and Roden, M.S. *Engineering Design for Electrical Engineers*, Prentice-Hall, 1990.
- [Williams 97] Williams, M.R. *A History of Computing Technology*, IEEE Computer Society, 1997.
- [Wirth 71a] Wirth N., *Program Development by Stepwise Refinement*, Communications of the ACM, Vol. 14, No. 4, pp. 221-227. April 1971.
- [Wirth 71b] Wirth, N. *The programming language PASCAL*. Acta Informatica 1, 1(1971), 35-63.
- [Yourdon & Constantine 79] Yourdon, E., & Constantine L.L., *Structured Design*, Prentice-Hall 1979.