Synthesis-Based Software Architecture Design

Bedir Tekinerdoğan

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Tekinerdogan, Bedir

Synthesis-Based Software Architecture Design /
Bedir Tekinerdogan. - [S.1.: s.n.]. -Ill.
PhD thesis University of Twente, Enschede, The Netherlands. - With index, -ref.
With summary in Dutch.
ISBN 90-365-1430-4
Subject headings: software architecture design/ synthesis / comparative analysis / philosophy / software engineering /mature engineering/ design algebra / object-oriented programming / alternative space analysis.

ISBN 90-365-1430-4

Printed by Print Partners Ipskamp, Enschede, The Netherlands.

SYNTHESIS-BASED SOFTWARE ARCHITECTURE DESIGN

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
Prof. dr. F.A. van Vught,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 23 maart 2000 te 15.00 uur.

door

Bedir Tekinerdogan

geboren op 15 maart 1970

te Besni, Adiyaman, Turkije

“There is one thing in this world which must never be forgotten.
If you were to forget everything else, but did not forget that,
then there would be no cause to worry;
whereas if you performed and remembered and did not forget every single thing,
but forgot that one thing,
then you would have done nothing whatsoever.”

- Rumi, Mesnevi

To the loving, caring and conscious people.…

Abstract

With the introduction of the first programming languages in the late 1940s and the early 1950s, software development has undergone several evolutionary changes, which provided opportunities for building larger and more complex software systems. This increased potentiality was soon followed by the realization that software is difficult to deliver on time, within the available budget and with the required quality factors such as reliability, stability and adaptability. To cope with this so-called *software crisis* an *engineering* approach to software development was proposed.

Many different attempts, ranging from improved programming languages to CASE tools, have been carried out during the last three decades to tackle the problems in software engineering that directly or indirectly lead to the symptoms of the software crisis. In the last decade, software architecture has gained a wide popularity as a fundamental concept in software engineering to support software quality factors. Software architecture embodies the overall structure of the system and likewise has a substantial impact on the quality aspects of the whole software system. Notwithstanding these various attempts developing high quality software systems still remains a difficult task.

To grasp the essence of software engineering and understand its inherent problems, this thesis provides a thorough and critical analysis of software engineering from a broad perspective. To this aim, software engineering is considered as a problem solving process whereby software solutions are produced for given technical problems. To explicitly reason about the concepts of problem solving, this thesis provides a model for problem solving that may be used for analyzing various problem-solving activities. In this thesis, this model is used for analyzing problem solving in software engineering and comparing it with the more mature problem solving disciplines of philosophy and traditional engineering, such as electrical engineering, mechanical engineering, civil engineering and chemical engineering. This conceptual and comparative analysis has resulted in a set of useful lessons and concepts that are essential for providing high-quality software but notably are missing in current software engineering practices.

A basic concept that is derived from our analysis process that may be essential for software engineering, is the concept of *synthesis*. Synthesis is a well-known problem solving process that is broadly and successfully applied in the traditional engineering disciplines. It includes explicit processes for technical problem analysis, solution domain analysis and alternative space analysis. In the technical problem analysis process, technical problems are identified and structured into loosely coupled sub-problems that are first independently solved and later integrated in the overall solution. In the solution domain analysis process, solution abstractions are extracted from the corresponding solution domains. In the alternative space analysis process different alternative solutions are searched and evaluated against explicit quality criteria.

In current software engineering practices the synthesis concept is not known and the three processes are not fully integrated. Since synthesis is a useful concept in mature problem solving it is worthwhile to integrate this in software engineering.

This thesis focuses on the software architecture design phase and attempts to improve the understanding on this subject by classifying and evaluating the current state-of-the-art software architecture design approaches. It appears that these approaches derive solution abstractions basically from the requirement specifications and the management of design alternatives is an implicit process. This causes a number of problems such as the difficulty in finding stable abstractions, difficulty in leveraging the architecture boundaries and poor semantics of the architectural components.

To address the problems of the state-of-the-art, the concept of synthesis is applied to software architecture design, resulting in a novel approach that we termed *synthesis-based software architecture design*. In this approach, the architectural abstractions are derived from the solution domains and the design alternatives are explicitly depicted and managed. This approach is illustrated with the design of an atomic transaction architecture for a distributed car dealer information system.

The architecture design can be realized by applying object-oriented analysis and design methods in which a set of heuristic rules are provided to guide software engineers to analyze, design and implement object-oriented software systems. This thesis introduces a new formalism, called *design algebra*, which provides techniques for explicitly depicting the set of architecture implementation alternatives, prioritizing these alternatives and selecting these based on quality factors. The techniques represented by design algebra can be integrated with the current object-oriented analysis and design methods and have been implemented as a set of tools.

Prologue

"By keenly confronting the enigmas that surround us, and by considering and analyzing the observations that I have made, I ended up in the domain of mathematics. Although I am absolutely without training in the exact sciences, I often seem to have more in common with mathematicians than with my fellow artists."
- M.C. Escher

The illustration on the cover page and the illustrations at the beginning of each chapter have been adopted from the works of Maurits Cornelis Escher (1898-1972), a Dutch graphic artist, who is most recognized for spatial illusions, impossible buildings and repeating geometric patterns. Escher began his studies at the School for Architecture and Decorative Arts in Haarlem but later shifted his emphasis to graphic arts whereby he mastered graphic and woodcutting techniques. Upon completion of his education, he traveled extensively through Southern France, Spain and Italy where he collected many inspirations for his work.

Many of his little-known early works tended toward realistic portrayals of the landscape and architecture observed during his travels. He began to turn away from realism towards the ideas which would make him famous when he visited for several times the 14th century Alhambra, a former Muslim palace in Granada, Spain. There he viewed the tile patterns that filled the entire space on the walls of Alhambra and spent many days sketching these tilings. Later he claimed that this "was the richest source of inspiration that I have ever tapped." This inspiration laid the foundation for his work after 1937. Replacing the abstract patterns of the tiles in Alhambra with recognizable figures, in the late 1930s Escher developed unique tessellations, that are regular arrangements of closed shapes that completely cover the plane without overlapping and without leaving any gaps. The artist also used this concept in creating his metamorphosis prints—one shape or object turning into something completely different. Escher also increasingly explored complex architectural mazes involving perspectival games and the representation of impossible constructions and spaces. As his work developed, he drew great inspiration from mathematical concepts and through his miraculous creations, Escher was able to put a symbolic bridge between the realms of art and science.

The reason why we have included Escher's illustrations in this thesis go beyond their ornamental meanings and form also an interesting and proper analogy for the presented work in this thesis. The analogy can be made from two perspectives. Firstly, Escher's works on impossible buildings, which are not realizable in the physical world, show a nice similarity with the way software architectures are developed today. The current software architecture design approaches have difficulties in identifying the right abstractions of software architectures, which often results in anomalous software architectures that are actually, similar to Escher's impossible buildings, difficult to realize. For each illustration at the beginning of each chapter, we have described the analogy between the illustration and software architectures in more detail. A second analogy that we can derive is the person and attitude of Escher himself. Escher adopted a humble approach and let himself inspire from other cultures and disciplines that finally resulted in his marvelous work that made him famous. The work in this thesis is the result of the adopted broad perspective of software engineering and the consideration of other mature disciplines that provided useful inspirations and lessons for solving the identified problems.

Currently, it is generally accepted that software engineering is still in its pre-mature phase and that it has to overcome many obstacles to become a mature engineering discipline. Despite of this consideration, up until today, researchers in software engineering have not put much serious effort in considering and learning useful lessons from the other mature disciplines. In this thesis we show that this is possible and describe the integration of the synthesis concept of traditional engineering to software engineering. I belief, though, that this is only a relatively small part of the many concepts and lessons that we may derive. To save time and effort we need to extent our inspiration sources in software engineering and provide a broader and integral view to solve our problems that have been already solved in some other disciplines.

Acknowledgements

A journey is easier when you travel together. Interdependence is certainly more valuable than independence. This thesis is the result of four and half years of work whereby I have been accompanied and supported by many people. It is a pleasant aspect that I have now the opportunity to express my gratitude for all of them.

The first person I would like to thank is my direct supervisor Mehmet Aksit. I have been in his project since 1993 when I started my MSc assignment. During these years I have known Mehmet as a sympathetic and principle-centered person. His overly enthusiasm and integral view on research and his mission for providing 'only high-quality work and not less', has made a deep impression on me. I owe him lots of gratitude for having me shown this way of research. He could not even realize how much I have learned from him. Besides of being an excellent supervisor, Mehmet was as close as a relative and a good friend to me. I am really glad that I have come to get know Mehmet Aksit in my life.

I would like to thank my promotor Anton Nijholt who kept an eye on the progress of my work and always was available when I needed his advises. I would also like to thank the other members of my PhD committee who monitored my work and took effort in reading and providing me with valuable comments on earlier versions of this thesis: Peter Apers, Ulrich Eisenecker, Dieter Hammer, Eddy Michiels, Ignas Niemegeers, Calton Pu and Motoshi Saeki. I thank you all.

My colleagues of the TRESE project all gave me the feeling of being at home at work. Klaas van den Berg, Lodewijk Bergmans, Pim van den Broek, Marc Evers, Joke Lammerink, Ali Noutash, Arend Rensink and Richard van de Stadt, many thanks for being your colleague.

The TRESE group also substantially contributed to the development of this work. Especially the strict and extensive comments and the many discussions and the interactions with Mehmet Aksit had a direct impact on the final form and quality of this thesis. Mehmet was even available on his vacation in Turkey from where he provided valuable comments on the thesis. I would like to thank Lodewijk, for our many discussions and providing me brotherly advises and tips that helped me a lot in staying at the right track. I thank Marc, for having shared the same room with me and for being a good colleague during the last two years of my PhD period. Lodewijk, Marc and Arend have read parts of my thesis and provided me valuable comments. Arend helped me much in getting things formal in a correct way. Pim helped me in finding the right algorithms that were needed for the tools that I implemented for chapter 5. Klaas helped me in finding the necessary literature.

Richard helpfully scanned the Escher illustrations. I would also like to thank Francesco Marcelloni, who visited us for a year, and with whom I had many discussions on the realm of science and life.

I had the pleasure to supervise and work with several students who did their graduation work in our project and have been somehow beneficial for the presented work in this thesis. Ferenc Schopbarteld provided a performance analysis of the various concurrency control techniques and provided a simulation environment. The results of the various simulations have been used in chapter 5 of the thesis. The synthesis approach that is presented in chapter 4 has matured in our project over the years through preliminary experimental approaches in which architectures for various applications were developed in MSc projects. Charles Vuijst designed an architecture for Image Algebra framework, Ronald Willems designed an architecture for Insurance Systems and Erik van der Arend designed an architecture for Quality Management Systems.

This research has been supported and funded by various organizations including Siemens-Nixdorf Software Center, The Dutch Ministry of Economical affairs under the SENTER program, The Dutch Organization for Scientific Research (NWO, 'Inconsistency management in the requirements analysis phase' project) and the AMIDST project of the Telematics Institute, University of Twente. I thank them all for their confidence in me. I am also grateful for the department of computer science at the University of Twente for providing me an excellent work environment during the past years and Charlotte Bijron, Alice Hoogvliet and Joke Lammerink for their cheerful assistance.

I am grateful for Fred Ahsmann and Andre van Dalen from Siemens Nixdorf who helped me to understand the NEDIS car dealer information system. It was a pleasure to work with them and the other employees of Siemens Nixdorf.

I feel a deep sense of gratitude for my late father and mother who formed part of my vision and taught me the good things that really matter in life. The happy memory of my father still provides a persistent inspiration for my journey in this life. I am grateful for my five brothers Mehmet, Ismail, Ebubekir, Ramazan and Haciali, for rendering me the sense and the value of brotherhood. I am glad to be one of them.

In my opinion, doing a PhD is a sacred task and this was definitely one of the best decisions of my life. Additional energy and vitality for this research was provided externally through my involvement in several social activities. Thereby, the Twente Association of Turkish Academics that I have served since its foundation in 1994 has played a substantial role and provided me the opportunity to meet many good friends. Ahmet, Ahmet Suat, Ata, Ayhan, Bekir, Burhan, Cetin, Faruk, Hamad, Latif, Mahmut, Mehmet, Metin, Murat, Nur, Orhan, Osman, Rachid, Ramazan, Said, Selim and Yasemin, I thank you all for having shared many experiences and thoughts with me throughout the last years. A special thanks goes to Osman Özturk, whom I have known for more than ten years now and who showed to be a kind, mostly helpful and trustful friend.

The principle, to see the good in everything and be a constructive part of the whole, made me feel responsible to participate in 'Broodje Religie', a dialogue group of people with different religions who aim to understand each other and grasp the values that are universal to mankind. These discussions helped me to keep the universal spirit that is essential for both religion and scientific research. Lots of thanks goes to Kees Kuyvenhoven and Helene van den Bempt who happily moderated the discussions of this group.

I am very grateful for my wife Leyla, for her love and patience during the PhD period. One of the best experiences that we lived through in this period was the birth of our son Irfan Davud, who provided an additional and joyful dimension to our life mission.

The chain of my gratitude would be definitely incomplete if I would forget to thank the first cause of this chain, using Aristotle's words, The Prime Mover. My deepest and sincere gratitude for inspiring and guiding this humble being.

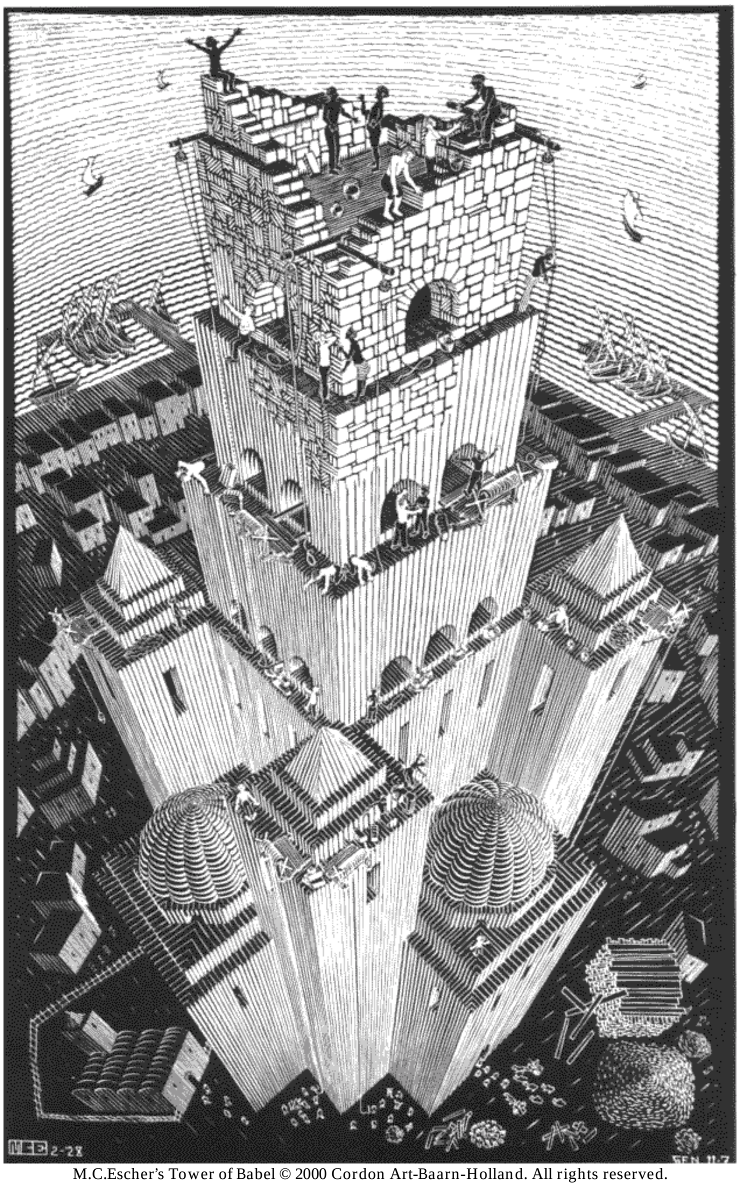
Bedir Tekinerdogan,

March 23, 2000
Enschede, The Netherlands

Contents

Chapter **1**

Introduction


M.C.Escher's Tower of Babel © 2000 Cordon Art-Baarn-Holland. All rights reserved.

M.C. Escher - Tower of Babel

The illustration on the previous page represents the Tower of Babel. According to the Quran (28:38) and the Bible (Genesis 11:1-9), a tower was erected by people in Babylonia with the intention to reach to heaven and God. Their attempts, however, failed because God interrupted the construction by causing among them a previously unknown confusion of languages and scattered them over the face of the earth.

Software Architecture Design Analogy

Software projects may attempt to build very large architectures to comprise a very broad set of alternative systems. This goal may be unrealizable and the software architecture may not be comprehensible for the many different stakeholders of the software architecture. The stakeholders may 'speak' different languages and as such be scattered which may easily lead to an unfinished software architecture.

1.1 Introduction

"Begin with the end in Mind."
- Stephen Covey, The 7 habits of highly effective people

The introduction of the first digital computers in the 1940s may be considered as the initiation of the history of software development. In these early days, the first software programs were written in machine language and were basically developed for numerical calculations in military projects. Later, with the introduction of the first programming languages in the late 1940s and the early 1950s, software development has undergone several evolutionary changes, which provided opportunities for building larger and more complex software intensive systems. This increased potentiality was soon followed by the realization that software was difficult to deliver on time, within the available budget and with the required quality factors such as reliability, stability, and adaptability. To cope with this so-called *software crisis* an *engineering* approach to software development was proposed at the NATO conference on software engineering in 1968. The aspirational term of *engineering* implied that software development should be based on the conceptual foundations on which the other engineering disciplines are relying.

Many different attempts have been carried out during the last three decades to tackle the problems in software engineering that directly or indirectly lead to the symptoms of the software crisis. These attempts have focused on many different fields, such as improved programming languages, improved modeling techniques, introduction of analysis and design methods, formal specifications, CASE tools etc. Notwithstanding the various attempts developing high quality software systems still remains a difficult task [Neumann 95][Gibbs 94].

1.2 Problem Statement

The alarming awareness that the symptoms of the software crisis seem to have persisted over the last decades may lead to the misleading opinion that *engineering* is not the right way for software development and other paradigms needs to be searched instead. But can software development as practiced today be legitimately considered as an engineering discipline? What are the basic elements of an engineering discipline and how are these manifested in current software development practices? How are the problems solved in traditional engineering disciplines such as electrical engineering, mechanical engineering, civil engineering and chemical engineering that have been matured over many centuries? Can software engineering learn useful lessons from these disciplines and are there concepts that may be useful for software engineering but are missing today?

Software developers or software *engineers* as they often call themselves seem not to take serious effort in learning from the traditional engineering disciplines. It appears that the same problems are recurring over time and in different projects and the different attempts are repeating the mistakes. To save time, effort and tremendous costs it is wise to learn from the experiences of the other disciplines who have already since long learned how to effectively solve problems.

An important development in the last decade is the introduction of software architecture as a fundamental concept in software engineering to support the required quality factors. Software architecture embodies the overall structure of the system and likewise has a substantial impact on the quality aspects of the whole software system. Currently, several architecture design approaches have been introduced and it would be worthwhile to investigate these architecture design approaches and identify their problems and see how to improve these approaches based on the necessary engineering concepts derived from the traditional engineering disciplines.

Various different implementation alternatives may be derived from the same conceptual architecture. Each alternative may have different adaptability, performance and reusability characteristics. Software is, however, rarely designed for optimal quality but rather it is a compromise of multiple considerations. The architecture design may be realized by applying object-oriented analysis and design methods in which a set of heuristic rules are provided to guide software engineers to analyze, design and implement object-oriented software systems. Current object-oriented analysis and design methods do not provide explicit means for depicting the set of alternatives, prioritizing these alternatives and balancing them based on several quality factors such as adaptability, reusability and performance. To solve this problem we may derive concepts from the traditional engineering disciplines and observe how they manage the various design alternatives.

1.3 The Approach

To grasp the essence of software engineering and understand its inherent problems, this thesis analyzes and evaluates software engineering from a problem solving perspective. For this purpose a model for problem solving is presented that allows to explicitly reason about software engineering from the problem solving concepts. The problem solving model is utilized for comparing software engineering with the more mature problem-solving disciplines of philosophy and traditional engineering disciplines. The comparative analysis of these disciplines is performed both to derive lessons from the past and of today practices.

To improve the understanding on the current state-of-the-art of software architectures, the current architecture design approaches are classified and evaluated and the fundamental problems are identified. The identification of the problems is followed by the integration of the concept of *synthesis* from the mature engineering disciplines to the software architecture design approach, resulting in a novel approach that is termed as *synthesis-based software architecture design*. *Synthesis* is a well-known concept in traditional engineering disciplines and involves the construction of sub-solutions for distinct loosely coupled sub-problems and the integration of these sub-solutions into a complete solution. During the synthesis process, design alternatives are searched and selected based on the existing solution domain knowledge. The synthesis-based software architecture design approach will be illustrated by applying it to the design of an atomic transaction system architecture for a distributed car dealer information system.

To cope with the various architecture implementation alternatives, a new formalism called *design algebra* is introduced. Design algebra is used to depict the space of design alternatives, and define design rules for comparing, evaluating and composing them. The techniques provided by design algebra can be integrated with object-oriented design methods. The applicability of Design Algebra is illustrated using the atomic transaction system example.

1.4 Contributions

This thesis provides a number of contributions that are described in the following.

1. *A conceptual model for problem solving to reason about various problem solving activities*

In chapter 2, a problem solving model to reason about problem solving activities is presented. Problem solving has been extensively studied in cognitive sciences such as [Smith & Browne 93],[Agre 82],[Rubinstein & Pfeiffer 80] and [Newell & Simon 76]. These studies are basically concerned with problem solving from a psychological perspective and attempt to understand the human thinking processes. Moreover, they usually do not explicitly consider the control aspects in problem solving. The presented problem solving model in chapter 2, provides a conceptual model that uniquely integrates the concepts of control, problem solving and context in one model. In addition, this model is not a cognitive model but rather defines the separate product concepts involved in problem solving and control that are suitable for expressing and reasoning about the engineering problem solving process.

2. *A broad perspective of software engineering for understanding its concepts*

Several publications have been written on the notion of software engineering and the software crisis and it is often claimed that software engineering is different from traditional engineering because it has particular and inherent complexities that are not present in other traditional engineering disciplines. Most of these studies, however, lack to view software engineering from a broad perspective and do not attempt to derive lessons from other mature problem solving disciplines. This thesis provides a broad perspective of software engineering and derives the common concepts of mature problem solving disciplines. For this, in chapter 2, a historical analysis of philosophy, traditional engineering and software engineering is presented.

3. *Classification and Evaluation of the state-of-the-art architecture design approaches*

In chapter 3, a classification and evaluation of the state-of-the-art architecture design approaches is presented. The novel classification can be used to characterize existing architecture design approaches. The evaluation results in the identification of the corresponding problems of each category of architecture design approach. This may help in improving the architecture design approaches.

4. *Synthesis-Based Software Architecture Design Approach*

Chapter 4 introduces the *Synthesis-Based Software Architecture Design Approach* that provides appropriate solutions for the identified problems of the state-of-the-art architecture design approaches. The novelty of this approach is that it integrates the explicit processes of technical problem analysis, solution domain analysis and alternative space analysis.

5. *Atomic Transaction System Architecture*

In chapter 4, the synthesis-based software aarchitecture design approach is illustrated for the design of the architecture of an atomic transaction system. The atomic transactions architecture provides abstractions of different transaction concepts such as transaction management, concurrency control and recovery management. As such the transaction architecture has a value by its own and can be used for the design of, for example, distributed systems.

6. *Design Algebra, providing techniques for alternative space management*

In chapter 5, the formalism *design algebra* that provides techniques for the management of design alternatives is introduced. For a given problem using design algebra the set of alternatives can be depicted, the alternatives prioritized with respect to quality factors and the appropriate alternatives may be selected. These techniques can be utilized and integrated in the current object-oriented design approaches. In section 5.4 we have provided a process for deriving object-oriented design alternatives that need to be balanced with respect to the adaptability quality factor. This process by its own may be of value for the software engineers because often adaptability is considered as an important quality factor in object-oriented design, though no explicit means are available for this.

7. Automation of design algebra techniques in a set of tools

In section 5.6 the *Rumi* environment is presented that implements the design algebra techniques as a set of tools. This CASE tool can be used to support the software engineer in depicting design spaces, reducing design spaces, prioritizing and selecting design alternatives.

1.5 Outline of the Thesis

Figure 1.1 represents the roadmap to the thesis. The rounded rectangles represent the chapters of the thesis, the arrows represent the relation between the chapters.

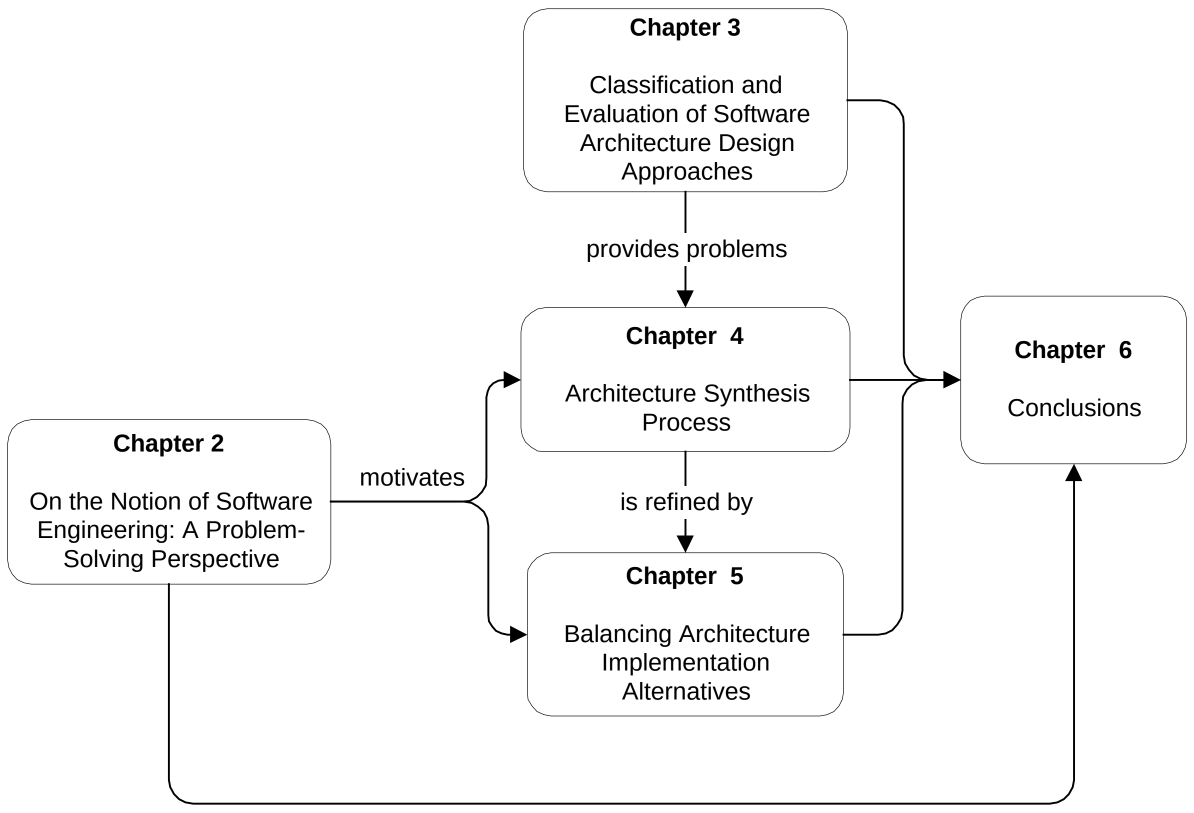

Chapter 3
Classification and Evaluation of Software Architecture Design Approaches
provides problems
Chapter 4
Architecture Synthesis Process
is refined by
Chapter 5
Balancing Architecture Implementation Alternatives
Chapter 2
On the Notion of Software Engineering: A Problem-Solving Perspective
motivates
Chapter 6
Conclusions

***Figure 1.1** Roadmap to the thesis*

Chapter 2 presents an analysis of the notion of software engineering based on a problem solving perspective. It provides an in-depth comparative analysis of software engineering with the more mature problem-solving disciplines of philosophy and the traditional engineering disciplines. The results of this study represent the motivation for chapter 4 and chapter 5. Chapter 4 addresses the problems of chapter 3 and provides a solution for these problems with the introduction of a novel architecture design approach, *synthesis-based software architecture design*. Chapter 5 refines the architecture design approach and presents a formalism, *design algebra*, for coping with architecture implementation alternatives. Finally, chapter 6 provides the solutions that are derived from the chapters 2, 3,4 and 5.

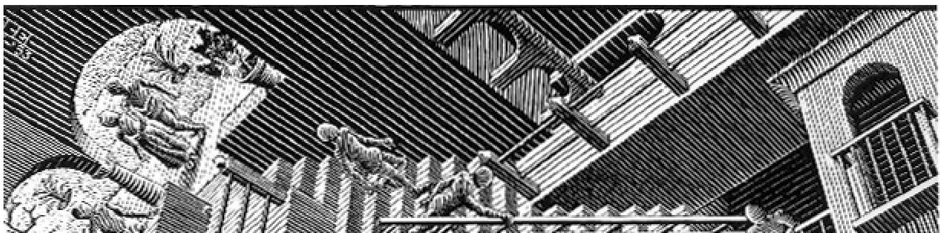
This thesis may be of relevance for software engineers and engineers of other disciplines.

Software engineers may use the presented comprehensive analysis of the notion of software engineering and improve the understanding on the basic concepts and identify the fundamental problems of software engineering (chapter 2). The classification and the evaluation of the current software architecture design approaches may help the software engineer to position and validate the architecture design approaches (chapter 3). The synthesis-based software architecture design approach can be applied in practice to define stable and adaptable software architectures (chapter 4). The atomic transaction architecture that is thoroughly described in chapter 4 may be applied in the design of distributed systems. Finally, the design algebra techniques of chapter 5 can be used in object-oriented analysis and design methods to depict the space of alternatives, prioritize these and balance them according to various quality factors.

Engineers of traditional engineering may be interested in the presented problem-solving model and the related comparative analysis in chapter 2. They may identify the explicit concepts of engineering and analyze, position and validate their own corresponding engineering discipline. In addition they may utilize the techniques of design algebra, that represents a formalism of the alternative space analysis in the traditional engineering disciplines.

Chapter 2

On the Notion of Software Engineering: A Problem Solving Perspective



M.C.Escher's Relativity © 2000 Cordon Art-Baarn-Holland. All rights reserved.

M.C. Escher - Relativity

In the illustration three forces of gravity are working perpendicularly to one another. Three earth-planes cut across each other at right angles, and human beings are living on each of them. It is impossible for the inhabitants of different worlds to walk or sit or stand on the same floor, because they have different perceptions of what is horizontal and what is vertical. Yet they may well share the use of the same staircase. On the top staircase illustrated here, two people are moving side by side and in the same direction, and yet one of them is going downstairs and the other upstairs. Contact between them is out of the question because they live in different worlds and therefore can have no knowledge of each other's existence.

Software Architecture Design Analogy

In building software architectures different 'forces' may work perpendicularly to one another in defining the form of the architecture. Different stakeholders may be interested in the same architecture, though, for a different reason. They may have different perceptions of the software architecture aspects and likewise may impose different forces on the architecture. Although, they may need to use the same architecture contact between them is generally out of question because they live in different 'worlds' and usually have no knowledge of each other's existence.

2.1 Introduction

"A problem cannot be solved at the same level of consciousness as it was created."
-Albert Einstein

For the last three decades many attempts have been carried out to address the software crisis that was identified by the end of the 1960s. 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 high-quality software. The significant problems we may face cannot be solved at the same level as they were initiated. To grasp the essence of software engineering and identify the missing concepts a broad view of software engineering may be needed instead.

If we consider the various definitions and attributed meanings of engineering in the literature, it follows that engineering essentially aims to provide an engineering solution 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. In this sense, engineering can be considered as a problem solving process, and to understand engineering it is necessary that we understand problem solving.

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 seeks to formulate problems accurately and attempts to find solutions for these problems is philosophy. The primary goal of philosophy is to understand the nature of things and as such it attempts to identify and describe the essential concepts [Kolenda 74]. Engineering disciplines are intrinsically related to forming engineering specific concepts and in addition several concepts may be based on concepts from philosophy.

The novelty of this chapter is that it presents a broad and general view of software engineering in order to grasp its essence and to identify the concepts that are necessary but are not well-defined or even missing in current practices. For this, an in depth comparative analysis of software engineering with traditional engineering and philosophy is provided based on problem solving concepts. Because of the adopted broad view, in addition to the software engineers, the chapter 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 chapter is presented in Figure 2.1.

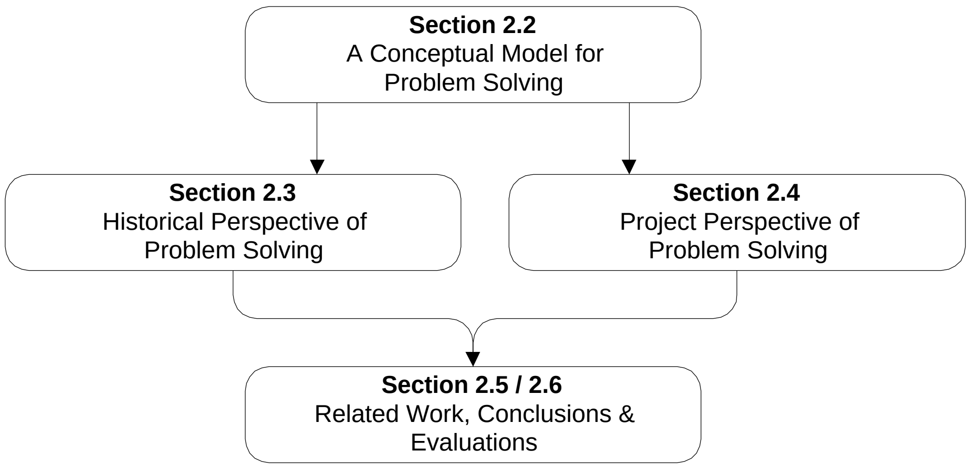

Section 2.2
A Conceptual Model for Problem Solving
Section 2.3
Historical Perspective of Problem Solving
Section 2.4
Project Perspective of Problem Solving
Section 2.5 / 2.6
Related Work, Conclusions & Evaluations

Figure 2.1 *The outline of the chapter*

In section 2.2 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 concepts. 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.

In section 2.3 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 so that the various problem solving concepts have evolved and matured over a much longer time. Studying the history of these mature disciplines will justify the problem solving model and allow to derive the concepts of value for current software engineering practices. In addition, 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.

Section 2.4 will analyze and evaluate problem solving in mature engineering and software engineering from a project perspective. For this purpose, a conceptual model for engineering that is derived from the previous problem solving model will be proposed. 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. The project perspective of software engineering will be compared with the project perspective of mature engineering to derive the missing concepts in the current software engineering paradigm.

In section 2.5 we will discuss the related work on problem solving and comparative analysis studies.

Finally, in section 2.6 we will give the evaluations and conclusions that include the fundamental conceptual problems of software engineering.

2.2 A Conceptual Model for Problem Solving

"Leave appearances. Come to essence and meaning.
Don't dwell in images or you will never mature."
- Yunus Emre

In this section we propose a conceptual model of problem solving for analyzing software engineering from a broad perspective. 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. Similar other studies have been carried out on problem solving from a psychological perspective and attempted to understand the human thinking processes [Smith & Browne 93]. In this chapter we are basically interested in a problem solving model that describes the separate identifiable concepts needed for understanding and expressing the concepts of engineering from a problem solving perspective. To this aim we will propose a problem solving model in section 2.2.1 and describe its basic concepts. In section 2.2.2 we will explain the purpose of the model.

2.2.1 The CPC Model

Figure 2.2 represents the model for problem solving that will be adopted in this chapter. 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 *C*ontrolled *P*roblem Solving in *C*ontext model, or the CPC model for short. Based on this assumption the model consists of three parts: *Problem Solving*, *Control* and *Context*. In the following, we will explain these parts in more detail.

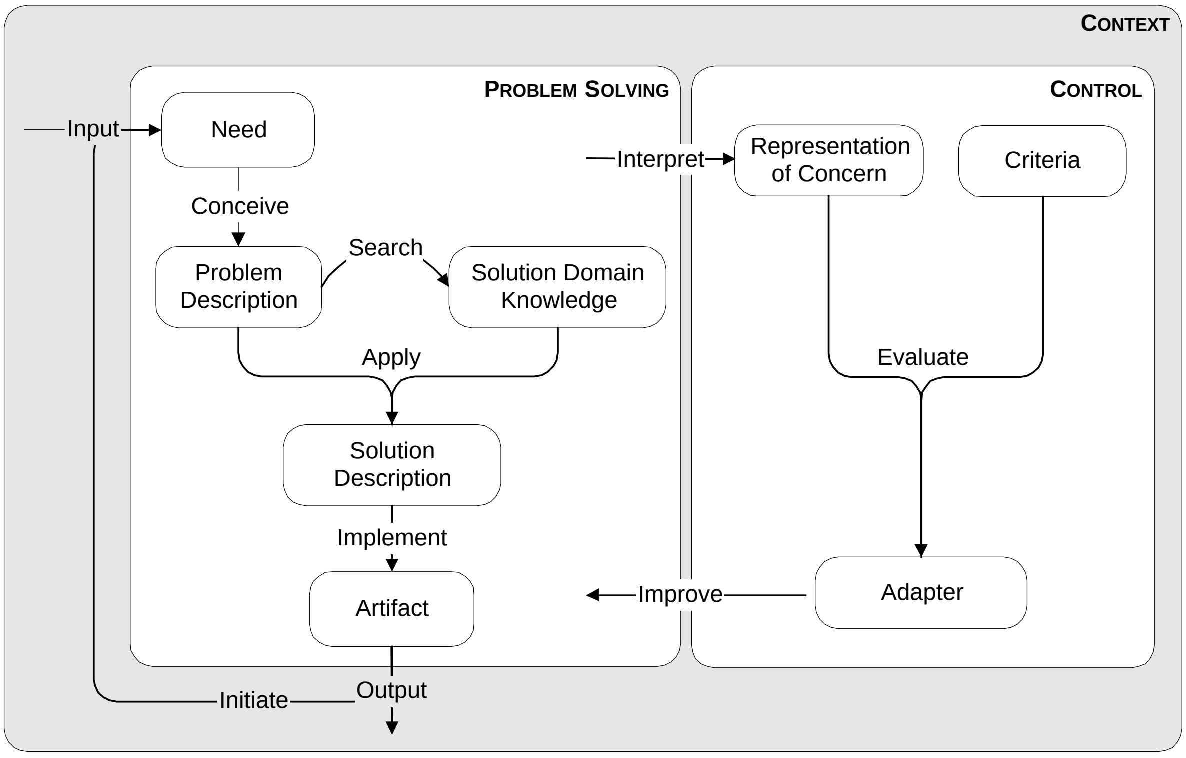


Figure 2.2 *The Controlled Problem solving in Context Model (CPC Model)*

Problem Solving

The problem-solving part consists of five 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.

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 criteria[1]. 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.*

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

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.

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 2.2. 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 2.2 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].

2.2.2 Purpose of the CPC Model

This CPC model serves as a basis for the whole chapter. 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 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 2.3 will mainly address the first two issues, that is, understanding the problem solving concepts and functions. Section 2.4 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 2.5 will address the last issue, that is, the conclusions and evaluations.

2.3 Historical Perspective of Problem Solving

Wer nicht von dreitausend Jahren sich weiss Rechenschaft zu geben,
bleibt im Dunkeln unerfahren, mag von Tag zu Tage leben.[2]
-Goethe

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 have 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 current 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

[2] From German: “Anybody who does not know about the history of the last three thousand years, will remain inexperienced in darkness and live from day to day".

software engineering. In this chapter we have chosen to study the historical perspective of problem solving in philosophy, mature engineering and software engineering.

In section 2.3.1 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 2.3.2 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 has 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 2.3.3 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.

2.3.1 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 parentheses.

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 explanations[3] 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*)[4]. They approached this problem by adhering to direct observations of the nature or by critical thought only (*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 irreducable 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 philosophies[5] took for granted that objective truth existed that could be discovered through a critical exchange of ideas by a community of thinkers.

We can explain the early history of philosophy using the CPC model. Problem solving was essentially almost a direct mapping from a problem to a solution. The control concepts can be explained in the following way. In 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 disciples 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)*.

[3] 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.

[4] The term “philosopher” was only later introduced by Heraclitus

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

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.

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 Sophists[6] 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)* that he affirmed basically 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 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)*[7].

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

[7] Socrates was later arrested of heresy and corrupting the young people. Finally, he was convicted and sentenced to death.

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. This conceptualization process formed a significant basis for the scientific developments in later centuries (*Output*).

From Solution Description to Implementation

With Aristotle the Greek political and social life broke down; Greek was involved in wars and first dominated by Persian 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)*.

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 degree[8] (*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 works[9] 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.

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 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

[8] 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.

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

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.

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. Positivism was founded by Auguste Comte in France and later developed in England by John Stuart Mill and Herbert Spencer. During the early 20th century, the *Logical Positivism* movement founded by Ludwig Wittgenstein, Bertrand Russel and George Edward 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. The materialistic development of the Positivism movement occurred in

Germany, where Karl Marx, the author of *Das Kapital* and the *Communist Manifesto* founded a new economic movement called *Socialism* (*Output*).

In his work *The Logic of Scientific Discovery* in 1934 [Popper 34] Popper criticized the prevailing view that scientific theories were developed through an inductive process in which the scientists induced theories from a set of observations. According to Popper scientific methods are fundamentally not inductive in character but rather hypothetico-deductive. This means that scientific theories are formed through hypotheses from which statements can be logically deduced. Popper proposed a criterion for testifiability or falsifiability for scientific validity. In case an experimental observation falsifies the theory it will be refuted, if the theory can explain the experimental observation it will continue to be tentatively accepted.

In 1962, Thomas Kuhn published *The Structure of Scientific Revolutions*, in which he argued that science is not a steady, cumulative acquisition of knowledge but instead is characterized as the successive transition from one *paradigm* to another through a process of revolution, which he termed as *paradigm shifts* [Kuhn 1962]. Kuhn defined the notion of *paradigm* basically as a collection of beliefs and agreements shared by the scientists of the community for understanding and solving problems.

Control of Problem Solving and Hermeneutics Philosophy

We will now focus on the function *Interpret*. In philosophy, interpretation concerns mainly that of the philosophical writings. During the Renaissance movement (*Context*), the need to reconstruct the original texts of classical thought and interpreting these, put a focus on the principles of interpretation (*Initiate)*. This problem was also identified in the realm of religion were 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 doctrine and as such broadened the authority of the interpretation of texts. The practical effect of this new view on interpretation led to a fragmentation of the Christianity into various religious groups (O*utput*).

Next to the problem on the authority for interpreting religious scriptures was the problem of *how* these 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. During the Reformation and Renaissance periods, *hermeneutic*[10] *philosophy* was established as an independent discipline. Hereby, it is 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*). Later, hermeneutics was developed by Heidegger, Dilthey, Gadamer, Vygotsky and Foucalt among others.

2.3.2 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.

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*).

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

[10] From Greek "hermeneuein" that means, "to interpret". This word itself is derived from the name of ancient Greek messenger god Hermes, who both delivered and explained the messages of the other gods.

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 melt 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, roads[11] and bridges were built *(Artifact*). To sustain plant growth, and thus the food production, irrigation was needed in places were 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 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.

[11] 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.

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*[12] [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.

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 capacity[13] of a single craftsman and it became very hard if not impossible to produce an artifact by a single person. Moreover, when many craftsmen were involved in the production, communication about the production process and the final artifact became important. A reflection on this process required a fundamental change in engineering problem solving. This initiated, especially in architecture, the necessity for drafting or designing (*Solution Description*), whereby the artifact is represented through a drawing before the actual production. Through drafting, engineers could communicate about the production of the artifact, evaluate the artifact before production and use the drafting or design as a guide for production. This enlightened the complexity of the engineering problems substantially. Currently, drafting plays an important role in all engineering disciplines.

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. There is a clear relation with the maturity

[12] 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.

[13] 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 (short term memory) is limited as well: it takes the mind about 5 seconds to accept a new chunk of information [Simon 62].

of the solution domain knowledge and the maturity of engineering. We will now therefore look at the concept *Solution Knowledge* in more detail.

Mathematical knowledge forms a principal basis for engineering disciplines and its application can be traced back in various civilizations throughout the history. In ancient Egypt, Sumer and Babylonia empirical geometry was adopted for land surveying and architecture (*Need*) [Nijholt & Ende 94]. 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* [Cooke 97].

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-Khawarizmi[14] 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.

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

[14] The word "algorithm" is derived from his name.

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 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 alcohol[15], 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.

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,

[15] 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.

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 motion of projectiles.

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

Specialization of Problem Solving Techniques

Chemical engineering and chemistry advanced in the 19th century. Through the development of electrochemistry and spectroscope many more chemical elements could be discovered. Mendelejev 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 Mendelejev 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 was 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. Quantum theory describes the nature of matter and energy on an atomic scale. The foundation for quantum theory was laid by Max Planck, who postulated in 1900 that energy can be emitted or absorbed only in discrete units called *quanta*. Later, 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 (*Initiate*, *Output*). 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. The theory of relativity developed by Einstein, describes the nature of matter and energy at a large scale, that is large velocity and/or mass, and shows the essential unity of matter and energy, and of space and time.

Development of Control and Automation

To extend the capacity of machines and humans and to control the engineering tasks more and more, automation became of interest. Automation is first applied in manufacture which required the *division of labor*, that is the decomposition of the manufacturing task into small independent steps, as it has been introduced in the latter half of the 18th Adam Smith in his book *An Inquiry into the Nature and Causes of the Wealth of Nations* (1776) . The next step necessary in the development of automation was *mechanization* that includes the application of machines that duplicated the motions of the worker. The advantage of automation was directly observable in the increased production efficiency.

Machines were built with automatic-control mechanisms that include a feedback control system providing the capacity for self-correction. The advent of the computer has greatly supported the use of feedback control systems in manufacturing processes. In modern industrial societies computers are used to support various engineering disciplines. Its broad application is in the support for drafting and manufacturing, that is, computer-aided design (CAD) and computer-aided manufacturing (CAM).

2.3.3 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.

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 [Nijholt & Ende 94]. 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. The ALGOL (Algorithm Language) compiler (1958) provided new concepts 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.

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 grew 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 and because of an immense marketing effort of the Sun company.

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 in the form of generative grammar models (*Solution Domain Knowledge*).

Knuth presented a comprehensive overview of a wide variety of algorithms and the analysis of them [Knuth 67]. 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] and even program families [Parnas 76].

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 [Szyperski 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.

2.4 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 2.4.1 will focus on mature engineering, section 2.4.2 will focus on software engineering. This study will enable us to position each engineering discipline and 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 commonalties and differences between mature engineering and software engineering.

2.4.1 Project Perspective of Problem Solving in Mature Engineering

"If I have seen further it is by standing on the shoulders of giants"
- Isaac Newton

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 2.3.

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 concept *Alternative(s)* is not explicit in the original CPC model but specific to the engineering model. In addition the concepts of the control part have been refined for engineering. We will explain these refinements in more detail in the following sections.

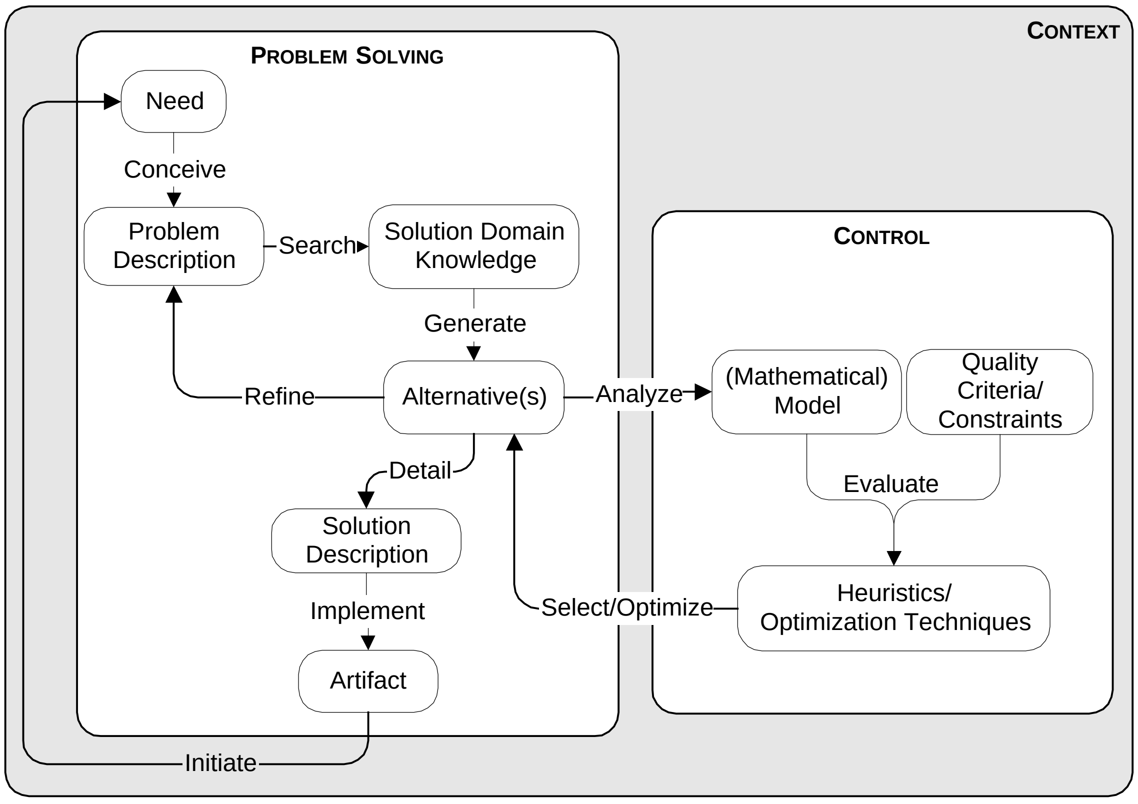


Figure 2.3 *A conceptual model of engineering based on the CPC model: CPC-Engineering model*

Conceiving Needs and Describing Problems

Every problem solving process starts with 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 representation 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 refers 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 discipline; 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 of the end-product for which different alternatives are defined and, for example, their economical cost may be calculated.

Synthesis

In mature engineering the process between the concept *Problem Description* to the concept *Solution Description* is termed *Synthesis*. In the following we will describe what kind of knowledge (*Solution Domain Knowledge*) is used in the synthesis process and how this is applied (*Apply*) to generate alternative solutions and evaluated.

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 from contributions of dozens of top experts in the field. Using the handbook, the engineer is guided with hundreds of valuable tables, charts, illustrations, formulas, equations, definitions, and appendices containing extensive conversion tables and usually sections covering mathematics. 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

In mature engineering alternatives are usually extracted 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.

Alternative generation is not a straightforward task and is considered as the most important and creative part of the synthesis process. Due to the complexity of the problem the number of steps to derive an alternative may become too large to explore with reasonable time and computational resources [Archer 65] and likewise engineering problems may be classified as NP-complete problems [Maimon & Braha 96].

Evaluation of Design Alternatives

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

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[16] 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.

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 discipline has its own specific artifact descriptions. In electrical engineering these are for example electrical circuit diagrams; in mechanical

[16] the number 190 represents the quality degree of ethanol

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.

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 2.4.

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.

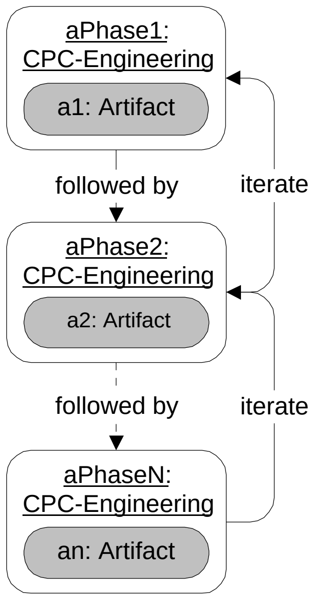


***Figure 2.4** The decomposition of the overall problem solving process into phases*

Since each phase is a problem solving process it adopts the concepts and functions as described by the engineering model in Figure 2.4 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 2.4 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 updating and/or recovering 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.

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 creative and routine design is very difficult to grasp 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.

2.4.2 Project Perspective of Problem Solving in Software Engineering

"To know others is wisdom, to know oneself is enlightenment."
- Lao Tsu, Tao Te Ching

We will now compare software engineering with the mature engineering disciplines. For software engineering we analyzed a selected set of textbooks [Sommerville 95], [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].

Conceiving Needs

In software engineering, the phase for conceiving the needs is referred to as *requirements analysis* which usually is started through an initial requirement specification of the client.

In mature engineering we have seen that the quality concept is already explicit in the problem description through the quantified objectives of the client. In software engineering this is quite different. 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.

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 that has developed over the last decades but in many aspects is not mature yet. 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, has been compiled in the book of Knuth [Knuth 67] and is 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. In that sense, the available handbooks of software engineering [Sommerville 95],[Pressman 94] are not comparable to the standard handbooks of mature engineering disciplines. Moreover, on many fundamental concepts in software engineering consensus among experts has still not been reached yet and research is ongoing.

In other engineering disciplines at phases when knowledge was lacking we observe that the basic attitude towards solving a problem was based on common sense, ingenuity and trial-and-error. 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, forms 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 requirement 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].

Alternative Generation

In designing software applications many alternatives have to be considered. The identification, selection, and balancing of these alternatives may be very difficult and 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 in contrast to 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 as 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.

2.5 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.

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 required[17].

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 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 chapter that lessons of value can be derived from other mature engineering disciplines. Petroski claims that lessons learned from failures can substantially advance engineering [Petroski 92]. In alignment with this, the paper [Holloway 99] derives practical hints for software development from two well-known 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

[17] This kind of motivation is similar to the philosophical thought of eclecticism that argued to adopt multiple thoughts rather than a single thought to be successful.

history of electrical engineering with the history of software engineering and thereby focuses on the failures in both engineering disciplines. According to Baber software development today is in a pre-mature phase analogous in many respects to the pre-mature phases of the now traditional engineering discipline that had also to cope with numerous failures[18]. Baber states that the fundamental causes of the failures in software development today are the same as the causes of the failures in electrical engineering 100 years ago, that is, lack of scientific mathematical knowledge or the failure to apply whatever such basis may exist. 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.

Several authors criticize the lack of well-designed experiments for measurement-based assessment in software engineering [Fenton et al. 94][Rombach et al. 93]. 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.

[18] Analyzing failures is a serious problem in all the engineering disciplines. Failures are hardly published or are hidden by the success stories because companies widely try to prevent circulating failure stories.

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.

2.6 Conclusion

The thesis of this chapter 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 the early history of philosophy, we have seen that 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 it 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 [Norman 98]. 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 addresses 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 of 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 any formal syntax will fail to completely determine its own interpretation and it should be rather grounded on the context that consists of the original meanings of the author and their relevance for the authors. This is to say that hermeneutics actually conforms to the 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 chapter 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 a 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. In software engineering the accumulation and formalization of reusable knowledge has only occurred for isolated domains such as algorithms, data structures and compiler techniques. It can be said that current software engineering practices depend more on creativity than it is the case in mature engineering disciplines.

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 are 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 requirement specification in several steps. The solution domain knowledge plays 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, heuristics are inherently necessary*

It appears that the synthesis in engineering processes is an NP-complete problem [Maimon & Braha 96][Kalay 87] 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.

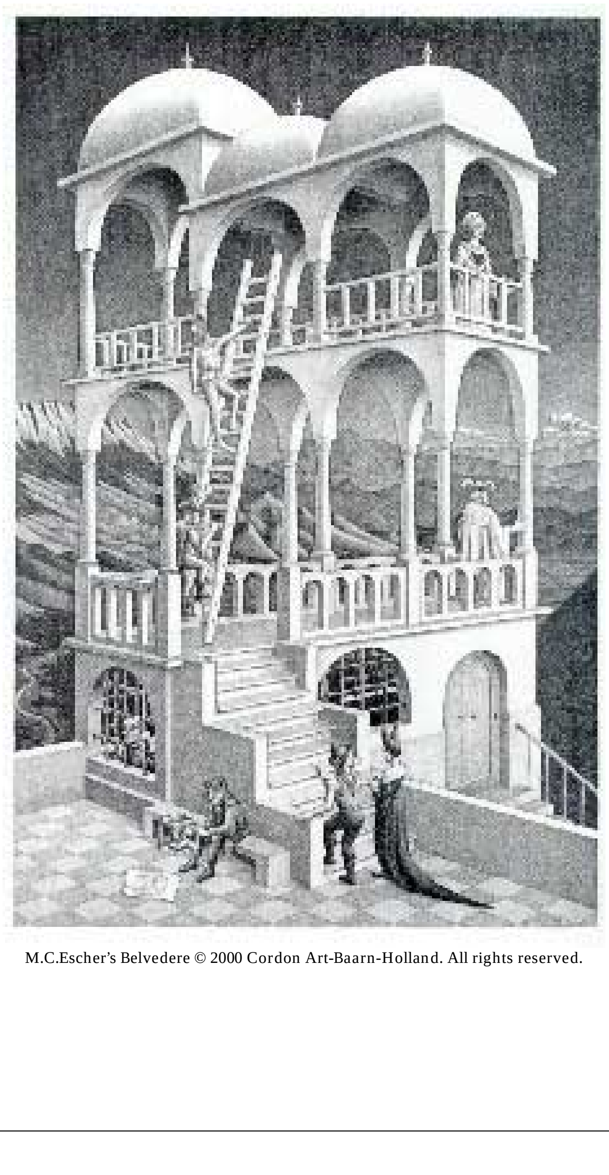
In addition to techniques for defining solution boundaries, mature engineering disciplines are supported with a set of mathematical optimization techniques to evaluate individual alternatives. In software engineering, the evaluation is largely based on heuristics or implicit criteria and suitable optimization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

Chapter 3

Classification and Evaluation of Software Architecture Design Approaches



M.C.Escher's Belvedere © 2000 Cordon Art-Baarn-Holland. All rights reserved.

M.C. Escher - Belvedere

In a three- dimensional world simultaneous front and back is an impossibility and so cannot be illustrated. Yet it is quite possible to draw an object which displays a differing reality when looked at from above and from below. The illustration on the previous page shows the illustration of a building that is physically difficult to realize. The lad sitting on the bench has got such a cube-like absurdity in his hands. He gazes thoughtfully at this incomprehensible object and seems unaware to the fact that the Belvedere behind him has been built in the same impossible style. On the floor of the lower platform, that is to say indoors, stands a ladder which two people are busy climbing. But as soon as they arrive at a floor higher, they are back in the open air and have to re-enter the building.

Software Architecture Design Analogy

It is useful to provide different perspectives of the same software architecture. Thereby, it is important that still the same 'reality' is represented. Unfortunately, due to inappropriate approaches and practices, the various perspectives of the same architecture may render different realities of the same architecture and lead to an impossible view of the overall software architecture. In addition, software engineers who criticize absurdities of architectures of other projects may not be aware of the impossible realities of their own developed software architecture.

3.1 Introduction

"Since in order to speak, one must first listen, learn to speak by listening."
- Rumi

Software architectures have gained a wide popularity in the last decade and is generally considered to play a fundamental role in coping with the inherent difficulties of the development of large-scale and complex software systems [Clements & Northrop 96]. A common assumption is that architecture design should support the required software system qualities such as robustness, adaptability, reusability and maintainability [Aksit et al. 00][Bass et al. 98]. Software architectures include the early design decisions and embody the overall structure that impacts the quality of the whole system. In the literature, hardly a consensus is reached yet for software architecture terminology, representations and architecture design approaches [Clements & Northrop 96] and several open problems have still to be solved. In this chapter we will focus on software architecture design approaches. For ensuring the quality factors it is generally agreed that, identifying the fundamental abstractions for architecture design is necessary. We maintain that the existing architecture design approaches have several difficulties in deriving the right architectural abstractions. To analyze, evaluate and identify the basic problems we will present a survey of the state-of-the-art architecture design approaches and motivate the obstacles in each approach.

The chapter is organized as follows. Section 3.2 provides a short background on software architectures in which existing definitions including our own definition of software architecture will be given. In section 3.3 a meta-model for software architecture design approaches will be given. This meta-model will serve as a basis for identifying the problems in our evaluation of architecture design approaches. In section 3.4 a classification, analysis and evaluation of the contemporary architectural approaches is presented. Finally, section 3.5 presents the conclusions and evaluations.

3.2 Notion of Software Architecture

In this section we focus on the meaning of software architecture by analyzing the prevailing definitions in section 3.2.1. In section 3.2.2 we provide our own definition that we consider as general and which covers the existing definitions.

3.2.1 Definitions

Software architectures are high-level design representations and facilitate the communication between different stakeholders, enable the effective partitioning and parallel development of the software system, provide a means for directing and evaluation, and finally provide opportunities for reuse [Bass et al. 98].

The term architecture is not new and has been used for centuries to denote the physical structure of an artifact [Webster 00]. The software engineering community has adopted the term to denote the gross-level structure of software-intensive systems. The importance of structure was already acknowledged early in the history of software engineering. The first software programs were written for numerical calculations using programming languages that supported mathematical expressions and later algorithms and abstract data types. Programs written at that time served mainly one purpose and were relatively simple compared to the current large-scale diverse software systems. Over time due to the increasing complexity and size of the applications, the global structure of the software system became an important issue [Shaw & Garlan 96]. Already in 1968, Dijkstra proposed the correct arrangement of the structure of software systems before simply programming [Dijkstra 68]. He introduced the notion of layered structure in operating systems, in which related programs were grouped into separate layers, communicating with groups of programs in adjacent layers. Later, Parnas maintained that the selected criteria for the decomposition of a system impact the structure of the programs and several design principles must be followed to provide a good structure [Parnas 72][Parnas 76]. Within the software engineering community, there is now an increasing consensus that the structure of software systems is important and several design principles must be followed to provide a good structure [Clements et al. 85].

In tandem with the increasing popularity of software architecture design many definitions of architecture have been introduced over the last decade, though, a consensus on a standard definition is still not established. We think that the reason why so many and various definitions on software architectures exist is because every author approaches a different perspective of the same concept of software architecture and likewise provides a definition from that perspective. Notwithstanding the numerous definitions it appears that the prevailing definitions do not generally conflict with each other and commonly agree that software architecture represents the gross-level structure of the software system consisting of components and relations among them [Bass et al. 98][19].

[19] Compare this to the parable of "the elephant in the dark", in which four persons are in a dark room feeling different parts of an elephant, and all believing that what they feel is the whole beast.

Looking back at the historical developments of architecture design we can conclude that similar to the many concepts in software engineering the concept of software architecture has also evolved over the years. We observe that this evolution took place at two fronts. First, existing stable concepts are specialized with new concepts providing a broader interpretation of the concept of software architecture. Second, existing interpretations on software architectures are abstracted and synthesized into new and improved interpretations. Let us explain this considering the development of the definitions in the last decade. The set of existing definitions is large and many other definitions have been collected in various publications such as [Soni et al. 95], [Perry & Wolf 92] and [SEI 00]. We provide only the definitions that we consider as representative.

> *"The logical and physical structure of a system, forged by all the strategic and tactical design decisions applied during development" [Booch 91]*

Hereby, software architecture represents a high-level structure of a software system. It is in alignment with the earlier concepts of software architecture as described by Dijkstra [Dijkstra 68] and Parnas [Parnas76]. The first variations of structure of architectures start to appear.

> *"We distinguish three different classes of architectural elements: processing elements; data elements; and connection elements. The processing elements are those components that supply the transformation on the data elements; the data elements are those that contain the information that is used and transformed; the connecting elements (which at times may be either processing or data elements, or both) are the glue that holds the different pieces of the architecture together. " [Perry & Wolf 92]*

This definition explicitly considers the interpretation on the elements of software architecture. It is a specialization of the previous architecture definitions and represents the functional aspects of the architecture focusing basically on the data-flow in the system.

> *" ...beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives. This is the software architecture level of design. " [Garlan & Shaw 93]*

This definition provides additional specializations of the structural issues.

"The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time."
[Garlan et al. 95]

This definition extends the previous definitions by including design information in the architectural specification.

"The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them." [Bass et al. 98]

This definition abstracts from the previous definitions and implies that software architectures have more than one structure and includes the behavior of the components as part of the architecture. The term component here is used as an abstraction of varying components [Bass et al. 98]. This definition may be considered as a sufficiently good representative of the latest abstraction of the concept of software architecture.

3.2.2 Architecture as a concept

The understanding on the concept of software architecture is increasing though there are still several shortcomings. Architectures consist of components and relations, but the term components may refer to subsystems, processes, software modules, hardware components or something else. Relations may refer to data flows, control flows, call-relations, part-of relations etc. To provide a consistent and overall definition on architectures, we need to provide an abstract yet a sufficiently precise meaning of the components and relations. For this we provide the following definition of architecture:

Architecture is a concept representing a set of abstractions and relations and constraints among these abstractions.

In essence this definition considers architecture as a *concept* that is general yet well-defined. We think that this definition is general enough to cover the various perspectives on architectures. To clarify this definition and discuss its implications we will provide a closer view on the notion of concept.

A *concept* is usually defined as a (mental) representation of a category of instances [Howard 87] and is formed by abstracting knowledge about instances. The process of assigning new instances to a concept is called *categorization* or *classification*. In this context, concepts are also called *categories* or *classes*. There are several theories on concepts and classification addressing the notions of concepts, classes, instances and categories [Lakoff 87][Smith & Medin 81][Parsons & Wand 97].

In the context of software architectures the architectural concepts are also abstractions of domain knowledge sources. The content of the domain sources, however, may vary per architecture design approach. We will elaborate on this topic in the following sections.

In general, three different views on the notion of concept are distinguished: the *classical view*, the *prototype view* and the *exemplar view*. The *classical view*[20] holds that all instances of the concept must share all the *defining* properties that are considered *necessary* and *sufficient* to define the concept. In other words, an instance must have all of the defining properties to be an instance of the concept and additionally, if an instance has at least the defining properties it is sufficient to denote it as an instance of the concept. In the *prototype view*[21] the concept is not described by *defining* properties but rather by *characterizing* properties, features that instances tend to have but need not to have. Basically the view proposes that a concept should be represented by some measure of central tendency of some instances, which is described by a *prototype*. A prototype is defined as an instance that has all the properties of the central tendency and as such is a highly typical instance or idealization. The *exemplar view* of concepts is quite different from the classical and the prototype view since hereby a concept does not represent an abstracted set of defining features or as a measure of a central tendency. The theory does not require abstraction of instances at all. Instead concepts are represented through *exemplars*. An exemplar is a specific instance of a certain category, which is used to represent the category.

Given the different views on concepts the question here is then which of the view of concepts is suitable for architectures. Basically, each view has its advantages and disadvantages and can be applied for solving a particular category of problems [Stillings et al. 95]. The classical view can be best applied for representing well-defined concepts. The prototype view and exemplar view on the contrary can be best applied in the early phases of concept formation in which specific instances are

[20] The classical view dates back to the philosophical works of Plato and Aristotle. Plato defined the notion of *forms*, which were defined as stable, immutable and ideal descriptions of things. Aristotle continued the research on classification and his work led to the classical view on categorization and concepts [Lakoff 87].

[21] The prototypical view has emerged from the philosophical treatments of Wittgenstein who maintained that for most concepts meaning is determined not by definition but by family resemblance [Wittgenstein 53].

discovered first and are later generalized. Accordingly, we may apply the prototypical and exemplar view in the early phases of architecture design and the classical view may be applied to define the stable architectural abstractions at later stages of the architecture design in which the knowledge on instances and concepts has got mature.

The definition has also implications for the structuring of concepts in software architectures. Two widely known structures are *taxonomies* and *partonomies* [Howard 87]. Taxonomies are usually represented by tree-like structures whereby the top-level concepts include the lower level concepts. Each taxonomy has both a horizontal and a vertical dimension. The vertical dimension represents the level of abstraction and the horizontal dimension represents mutually exclusive categories at the same abstraction level. A particular abstraction level that is called the *basic-level* defines the useful abstractions [Rosch et al. 76]. *Partonomies* define structures in which concepts are related to each other through part-whole relations rather than class inclusion. Taxonomies and partonomies are the basic well-known structures, however, other structuring mechanisms that include various relations between concepts, such as associations, may also be applied.

Since an architecture is a structure of concepts and each concept may represent structures themselves, the definition implies that an architecture may have different structures. In the simple case the architecture consist of a set of concepts that can be considered as 'atomic' and do not have an internal structure. For large software systems, however, it is necessary to define the architecture from various perspectives such as, for example, from a logical view, the process view, the development view and the physical view [Kruchten 95]. Therefore, at the highest abstraction level the software architecture may consist of concepts that each represents different architectural views.

Concepts are not just arbitrary abstractions or groupings of a set of instances but are defined by a consensus of experts in the corresponding domain. As such concepts are stable and well-defined abstractions with rich semantics. The definition thus enforces that each architecture consists of components that do not only represent arbitrary groupings or categories but are semantically well-defined.

3.3 Meta-Model for Architecture Design Approaches

In this section we provide a meta-model that is an abstraction of various architecture design approaches. We will use this model to analyze and compare current architecture design approaches, which we will describe in the subsequent section. The meta-model is given in Figure 3.1.

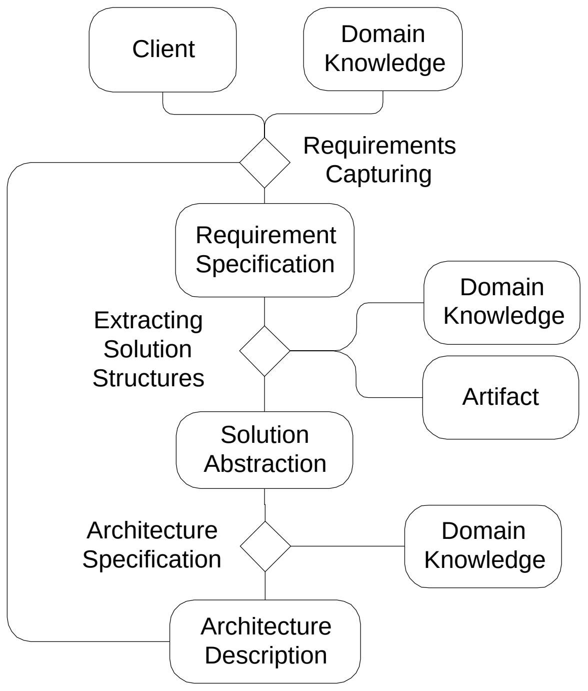


Figure 3.1 *Meta-model for architecture design approaches*

The rounded rectangles represent the concepts and the lines represent the association between these concepts. The diamond symbol represents an association relation between three or four concepts. Let us now describe the concepts individually.

The concept *Client* represents the stakeholder(s) who is/are interested in the development of a software architecture design. A stakeholder may be a customer, end-user, system developer, system maintainer, sales manager etc.

The concept *Domain Knowledge* represents the area of knowledge that is applied in solving a certain problem. We will elaborate on this concept in the next sub-section.

The concept *Requirement Specification* represents the specification that describes the requirements for the architecture to be developed.

The concept *Artifact* represents the artifact descriptions of a certain method. This is for example, the description of the artifact Class, Operation, Attribute, etc. In general each artifact has a related set of heuristics for identifying the corresponding artifact instances.

The concept *Solution Abstraction* defines the conceptual representation of a (sub)-structure of the architecture.

The concept *Architecture Description* defines a specification of the software architecture.

In Figure 3.1 there are two quaternary association relations and one ternary association relation.

The quaternary association relation called *Requirements Capturing* defines the association relations between the concepts *Client*, *Domain Knowledge*, *Requirement Specification* and *Architecture Description*. This association means that for defining a requirement specification the client, the domain knowledge and the (existing) architecture description can be utilized. The order of processing is not defined by this association and may differ per architecture design approach.

The quaternary association relation called *Extracting Solution Structures* is defined between the concepts *Requirement Specification, Domain Knowledge, Artifact* and *Solution Abstraction*. This describes the structural relations between these concepts to derive a suitable solution abstraction.

The ternary association relation *Architecture Specification* is defined between the concepts *Solution Abstraction*, *Architecture Description* and *Domain Knowledge* and represents the specification of the architecture utilizing these three concepts.

Various architecture design approaches can be described as instantiations of the meta-model in Figure 3.1. Each approach will differ in the ordering of the processes and the particular content of the concepts.

3.3.1 Domain Knowledge

In the meta-model the concept *Domain Knowledge* is used three times. Since this concept plays a fundamental role in various architectural design approaches we will now elaborate on this concept.

The term domain has different meanings in different approaches. We distinguish between the following specializations of this concept: *Problem Domain Knowledge, Business Domain Knowledge, Solution Domain Knowledge* and *General Knowledge*. This classification of domain knowledge concepts is shown in Figure 3.2.

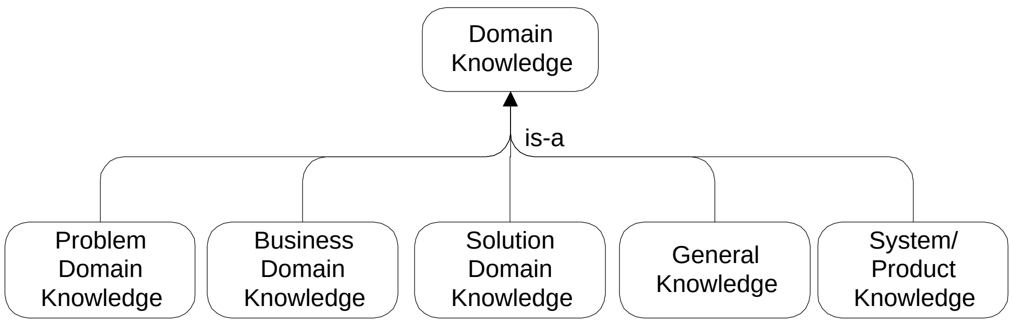

Domain Knowledge
is-a
Problem Domain Knowledge
Business Domain Knowledge
Solution Domain Knowledge
General Knowledge
System/ Product Knowledge

Figure 3.2 *Different specializations of the concept Domain Knowledge*

The concept *Problem Domain Knowledge* refers to the knowledge on the problem from a client's perspective. It includes requirement specification documents, interviews with clients, prototypes delivered by clients etc.

The concept *Business Domain Knowledge* refers to the knowledge on the problem from a business process perspective. It includes knowledge on the business processes and also customer surveys and market analysis reports.

The concept *Solution Domain Knowledge* refers to the knowledge that provides the domain concepts for solving the problem and which is separate from specific requirements and the knowledge on how to produce software systems from this solution domain. This kind of domain knowledge is included in for example textbooks, scientific journals, and manuals.

The concept *General Knowledge* refers to the general background and experiences of the software engineer and also may include general rules of thumb.

The concept *System/Product Knowledge* refers to the knowledge about a system, a family of systems or a product.

3.4 Analysis and Evaluation of Architecture Design Approaches

A number of approaches have been introduced to identify the architectural design abstractions. We classify these approaches as *artifact-driven*, *use-case-driven*, *domain-driven* and *pattern-driven* architecture design approaches. The criterion for this classification is based on the adopted basis for the identification of the key abstractions of architectures. Each approach will be explained as a realization of the meta-model described in Figure 3.1.

3.4.1 Artifact-driven Architecture Design

We term artifact-driven architecture design approaches as those approaches that extract the architecture description from the artifact descriptions of the method. Examples of artifact-driven architectural design approaches are the popular object-oriented analysis and design methods such as OMT [Rumbaugh et al. 91] and OAD [Booch 91]. A conceptual model for artifact-driven architectural design is presented in Figure 3.3. Hereby the labeled arrows represent the process order of the architectural design steps. The concepts *Analysis & Design Models* and *Subsystems* in Figure 3.3 together represent the concept *Solution Abstraction* of Figure 3.1. The concept *General Knowledge* represents a specialization of the concept *Knowledge Domain* in Figure 3.1.

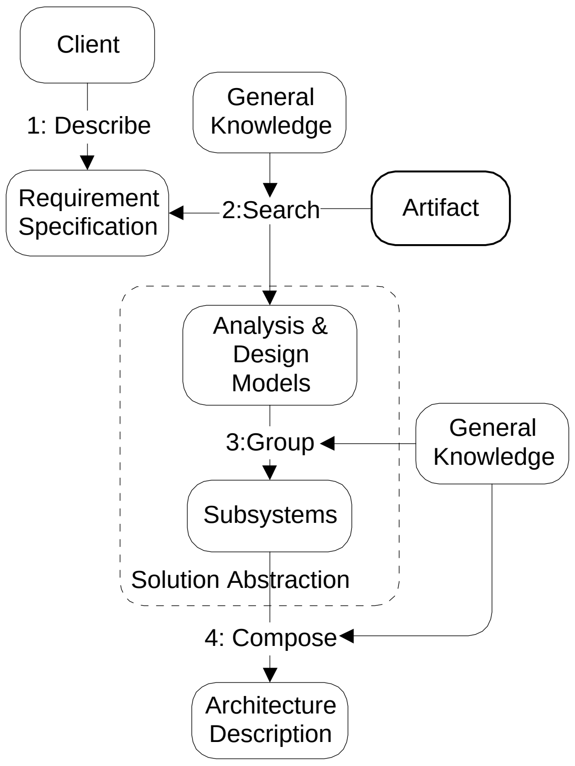


Figure 3.3 *Conceptual model of artifact-driven architectural design*

We will explain this model using OMT [Rumbaugh et al. 91], which can be considered as a suitable representative for this category. In OMT, architecture design is not an explicit phase in the software development process but rather an implicit part of the design phase. The OMT method [Rumbaugh et al. 91] consists basically of the phases *Analysis*, *System Design*, and *Object Design*. The arrow *1:Describe* represents the description of the requirement specification. The arrow *2:Search* represents the search for the artifacts such as classes in the requirement specification in the analysis phase. An example of a heuristic rule for identifying tentative class artifacts is the following:

IF an entity in the requirement specification is relevant
THEN select it as a Tentative Class.

The search process is supported by the general knowledge of the software engineer and the heuristic rules of the artifacts that form an important part of the method. The result of the *2:Search* function is a set of artifact instances that is represented by the concept *Analysis &Design Models* in Figure 3.3.

The method follows with the *System Design* phase that defines the overall architecture for the development of the global structure of a single software system by grouping the artifacts into *subsystems* [Rumbaugh et al. 91]. In Figure 3.3 this grouping function is represented by the function *3:Group*. The software architecture consists of a composition of subsystems, which is defined by the function *4:Compose* in Figure 3.3. This function is also supported by the concept *General Knowledge.*

Problems

In OMT the architectural abstractions are represented by grouping classes that are elicited from the requirement specification. We maintain that hereby it is difficult to extract the architectural abstractions. We will explain the problems using the example described in OMT on an Automated Teller Machine (ATM) which concerns the design of a banking network [Rumbaugh et al. 91]. Hereby, bank computers are connected with ATMs from which clients can withdraw money. In addition, banks can create accounts and money can be transferred and/or withdrawn from one account to another. It is further required that the system should have an appropriate recordkeeping and secure provisions. Concurrent accesses to the same account must be handled correctly.

The problems that we identified with respect to architecture development are as follows:

- *Textual requirements are imprecise, ambiguous or incomplete and are less useful as a source for deriving architectural abstractions*

In OMT artifacts are searched within the textual requirement specification and grouped into subsystems, which form the architectural components. Textual requirements, however, may be imprecise, ambiguous or incomplete and as such are not suitable as a source for identification of well-defined architectural abstractions. In the example, three subsystems are identified: *ATM Stations, Consortium Computer* and *Bank Computers*. These subsystems group the artifacts that were identified from the requirement specification. The example only includes one class artifact called *Transaction* since this was the only artifact that could be discovered in the textual requirement specification. Publications on transaction systems show that many concerns such as scheduling, recovery, deadlock management etc. are included in designing transaction systems [Elmagarmid 91][Date 90][Bernstein & Newcomer 97]. Therefore, we would expect additional classes that could not be identified from the requirement specification.

- *Subsystems have poor semantics to serve as architectural components*

In the given example, the component *ATM stations* represents a subsystem, that is, an architectural component. The subsystem concept serves basically as a grouping concept and as such has very poor semantics[22]. For the subsystem *ATM stations* it is

[22] In [Aksit & Bergmans 92] this problem has been termed as subsystem-object distinction.

for example not possible to define the architectural properties, architectural constraints with the other subsystems, and the dynamic behavior. This poor semantics of subsystems makes the architecture description less useful as a basis for the subsequent phases of the software development process.

- *Composition of subsystems is not well-supported*

Architectural components interact, coordinate, cooperate and are composed with other architectural components. OMT provides, however, no sufficient support for this process. In the given example, the subsystem *ATM Stations, Consortium Computer* and *Bank Computers* are composed together, though, the rationale for the presented structuring process is performed implicitly. One could provide several possibilities for composing the subsystems, though, the method lacks rigid guidelines for composing and specifying the interactions between the subsystems.

3.4.2 Use-Case driven Architecture Design

In the use-case driven architecture design approach *use cases* are used as the primary artifacts for deriving the architectural abstractions. A *use case* is defined as a sequence of actions that the system provides for *actors* [Jacobson et al. 99]. Actors represent external roles with which the system must interact. The actors and the use cases together form the use case model. The use case model is meant as a model of the system's intended functions and its environment, and serves as a contract between the customer and the developers. The Unified Process [Jacobson et al. 99] applies a use-case driven architecture design approach. The conceptual model for the use-case driven architecture design approach in the Unified Process is given in Figure 3.4. Hereby, the dashed rounded rectangles represent the concepts of Figure 3.1. For example the concepts *Informal Specification* and the *Use-Case Model* together form the concept *Requirement Specification* in Figure 3.1.

The Unified Process consists of *core workflows* that define the static content of the process and describe the process in terms of activities, workers and artifacts. The organization of the process over time is defined by phases. The Unified Process is composed of six core workflows: *Business Modeling, Requirements, Analysis, Design, Implementation* and *Test*. These core workflows result respectively in the following separate models: *business & domain model, use-case model, analysis model, design model, implementation model* and *test model.*

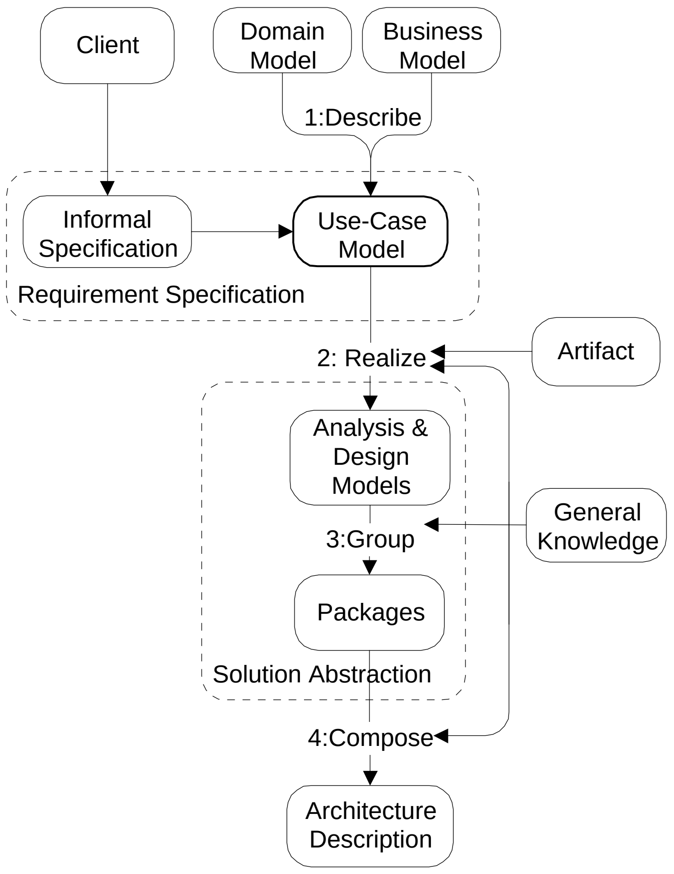


Figure 3.4 *Conceptual model of use-case driven architectural design*

In the requirements workflow, the client's requirements are captured as use cases which results in the use-case model. This process is defined by the function *1:Describe* in Figure 3.4. Together with the informal requirement specification, the use case model forms the requirement specification. The development of the use case model is supported by the concepts *Informal Specification*, *Domain Model* and *Business Model* that are required to set the system's context. The *Informal Specification* represents the textual requirement specification. The *Business Model* describes the business processes of an organization. The *Domain Model* describes the most important classes within the context of the domain. From the use case model the architecturally significant use cases are selected and *use-case realizations* are created as it is described by the function *2:Realize*. Use case realizations determine how the system internally performs the tasks in terms of collaborating objects and as such help to identify the artifacts such as classes. The use-case realizations are supported by the knowledge on the corresponding artifacts and the general knowledge. This is represented by the arrows directed from the concepts *Artifact* and *General Knowledge* respectively, to the function *2:Realize*. The output of this function is the concept *Analysis & Design Models*, which represents the identified artifacts after use-case realizations.

The analysis and design models are then grouped into *packages* which is represented by the function *3:Group*. The function *4:Compose* represents the definition of interfaces between these packages resulting in the concept *Architecture Description*. Both functions are supported by the concept *General Knowledge*.

Problems

In the Unified Process, first the business model and the domain model are developed for understanding the context. Use case models are then basically derived from the informal specification, the business model and the domain model. The architectural abstractions are derived from realizations of selected use cases from the use case models.

We think that this approach has to cope with several problems in identifying the architectural abstractions. We will motivate our statements using the example described in [Jacobson et al. 99, pp. 113] that concerns the design of an electronic banking system in which the internet will be used for trading of goods and services and likewise include sending orders, invoices, and payments between sellers and buyers. The problems that we encountered are listed as follows:

- *Leveraging detail of domain model and business model is difficult*

The business model and domain models are defined before the use case model. The question raises then how to leverage the detail of these models. Before use cases are known it is very difficult to answer this question since use cases actually define what is to be developed. In [Jacobson et al. 99 pp. 120] a domain model is given for an electronic banking system example. Domain models are derived from domain experts and informal requirement specifications. The resulting domain model includes four classes: *Order, Invoice, Item* and *Account*. The question here is whether these are the only important classes in electronic banking systems. Should we consider also the classes such as *Buyer* and *Seller*? The approach does not provide sufficient means for defining the right detail of the domain and business models[23].

- *Selecting architecturally relevant use-cases is not systematically supported*

For the architecture description, 'architecturally relevant' use cases are selected. The decision on which use cases are relevant lacks objective criteria and is merely dependent on some heuristics and the evaluation of the software engineer. For example, in the given banking system example, the use case *Withdraw Money* has

[23] Use cases focus on the functionality for each user of the system rather than just a set of functions that might be good to have. In that sense, use cases form a practical aid for leveraging the requirements.

been implicitly selected as architecturally relevant and other use cases such as *Deposit Money* and *Transfer between Accounts* have been left out.

- *Use-cases do not provide a solid basis for architectural abstractions*

After the relevant use cases have been selected they are *realized* which means that analysis and design classes are identified from the use cases. Use-case realizations are supported by the heuristic rules of the artifacts, such as classes, and the general knowledge of the software engineer. This is similar to the artifact-driven approach in which artifacts are discovered in the textual requirements. Although use cases are practical for understanding and representing the requirements, we maintain that they do not provide a solid basis for deriving architectural design abstractions. Use cases focus on the problem domain and the external behavior of the system. During use case realization transparent or hidden abstractions that are present in the solution domain and the internal system may be difficult to identify. Thus even if all the relevant use cases have been identified it may still be difficult to identify the architectural abstractions from the use case model. In the given banking system example, the use case-realization of *Withdraw Money* results in the identification of the four analysis classes *Dispenser*, *Cashier Interface*, *Withdrawal* and *Account* [Jacobson et al. 99, pp. 44]. The question here is whether these are all the classes that are concerned with withdrawal. For example, should we also consider classes such as *Card* and *Card Check*? The transparent classes cannot be identified easily if they have not been described in the use case descriptions.

- *Package construct has poor semantics to serve as an architectural component*

The analysis and design models are grouped into package constructs. Packages are, similar to subsystems in the artifact-driven approach, basically grouping mechanisms and as such have poor semantics. The grouping of analysis and design classes into packages and the composition of the packages into the final architecture are also not well supported and are basically dependent on the general knowledge of the software engineer. This may again lead to ill-defined boundaries of the architectural abstractions and their interactions.

3.4.3 Domain-driven Architecture Design

Domain-driven architecture design approaches derive the architectural design abstractions from domain models. The conceptual model for this domain-driven approach is presented in Figure 3.5.

Domain models are developed through a domain analysis phase represented by the function *2:Domain Analysis*. Domain analysis can be defined 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 & Arrango 91]. The function *2:Domain Analysis* takes as input the concepts *Requirement Specification* and *Domain Knowledge* and results in the concept *Domain Model.* Note that both the concepts *Solution Domain Knowledge* and *Domain Model* in Figure 3.5 represent the concept *Domain Knowledge* in the meta-model of Figure 3.1.

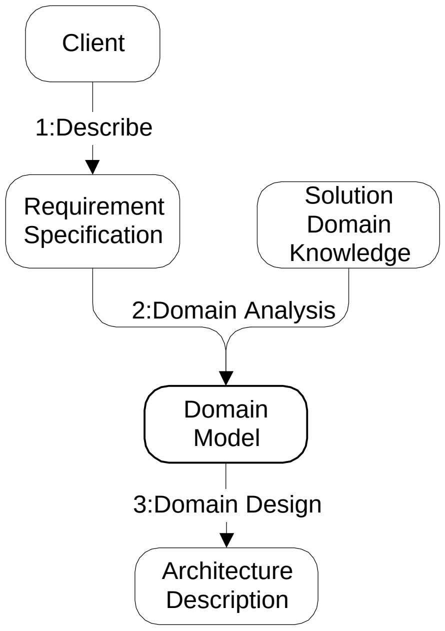


Figure 3.5 *Conceptual model for Domain-Driven Architecture Design*

The domain model may be represented using different representation forms such as classes, entity-relation diagrams, frames, semantics networks, and rules. Several *domain analysis* methods have been published, e.g. [Gomaa 92], [Kang et al. 90], [Prieto-Diaz & Arrango 91], [Simos et al. 96] and [Czarnecki 99]. Two surveys of various domain analysis methods can be found in [Arrango 94] and [Wartik & Prieto-Diaz 92]. In [Czarnecki 99] a more recent and extensive up-to-date overview of domain engineering methods is provided.

In this chapter we are mainly interested in the approaches that use the domain model to derive architectural abstractions. In Figure 3.5, this is represented by the function *3:Domain Design.* In the following we will consider two domain-driven approaches that derive the architectural design abstractions from domain models.

Product-line Architecture Design

In the product-line architecture design approach, an architecture is developed for a *software product-line* that is defined as a group of software-intensive products sharing a common, managed set of features that satisfy the needs of a selected market or mission area [Clements & Northrop 96]. A *software product line architecture* is an abstraction of the architecture of a related set of products. The product-line architecture design approach focuses primarily on the reuse within an organization and consists basically of *the core asset development* and *the product development*. The core asset base often includes the architecture, reusable software components, requirements, documentation and specification, performance models, schedules, budgets, and test plans and cases [Bass et al. 97a], [Bass et al. 97b], [Clements & Northrop 96]. The core asset base is used to generate or integrate products from a product line.

The conceptual model for product-line architecture design is given in Figure 3.6. The function *1:Domain Engineering* represents the core asset base development. The function *2:Application Engineering* represents the product development from the core asset base.

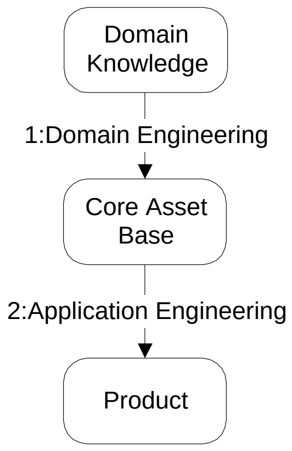


Figure 3.6 *A conceptual model for a Product-Line Architecture Design*

Note that various software architecture design approaches can be applied to provide a product-line architecture design. In the following section we will describe the DSSA approach that follows the conceptual model for product-line architecture design in Figure 3.6.

Domain Specific Software Architecture Design

The *domain-specific software architecture* (DSSA) [Hayes-Roth 94][Tracz & Coglianese 92] may be considered as a multi-system scope architecture, that is, it derives an architectural description for a family of systems rather than a single-system. The conceptual model of this approach is presented in Figure 3.7. The basic artifacts of a DSSA approach are the *domain model*, *reference requirements* and the *reference architecture*. The DSSA approach starts with a domain analysis phase on a set of applications with common problems or functions. The analysis is based on *scenarios* from which functional requirements, data flow and control flow information is derived. The *domain model* includes scenarios, domain dictionary, context (block) diagrams, ER diagrams, data flow models, state transition diagrams and object models.

In addition to the domain model, *reference requirements* are defined that include functional requirements, non-functional requirements, design requirements and implementation requirements and focus on the solution space. The domain model and the reference requirements are used to derive the *reference architecture*. The DSSA process makes an explicit distinction between a *reference architecture* and an *application architecture*. A reference architecture is defined as the architecture for a family of application systems, whereas an application architecture is defined as the architecture for a single system. The application architecture is instantiated or refined from the reference architecture. The process of instantiating / refining and / or extending a reference architecture is called *application engineering*.

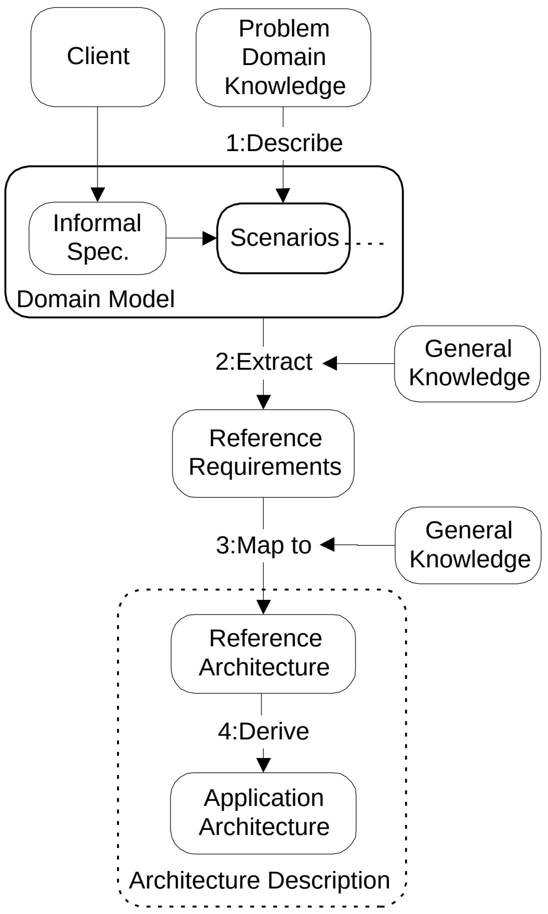


***Figure 3.7** Conceptual model for Domain Specific Software Architecture (DSSA) approach*

Problems

Since the term domain is interpreted differently there are various domain-driven architecture design approaches. We list the problems for problem domain analysis and solution domain analysis.

- *Problem domain analysis is less effective in deriving architectural abstractions*

Several domain-driven architecture approaches interpret the domain as a problem domain. The DSSA approach, for example, starts from an informal problem statement and derives the architectural abstractions from the domain model that is based on scenarios. Like use cases, scenarios focus on the problem domain and the external behavior of the system. We think that approaches that derive abstraction from the problem domain, such as the DSSA approach, are less effective in deriving the right architectural abstractions. Let us explain this using the example in [Tracz 95] in which an architecture for a theater ticket sales application is constructed using the DSSA approach. In this example a number of scenarios such as *Ticket Purchase, Ticket Return, Ticket Exchange, Ticket Sales Analysis,* and *Theater Configuration* are described and accordingly a domain model is defined based on these scenarios. The question hereby is whether the given scenarios fully describe the system and as such result in the right leverage of the domain model. Are all the important abstractions identified? Do there exist redundant abstractions? How can this be evaluated? Within this approach and generally approaches that derive the abstractions from the problem domain these questions remain rather unanswered.

- *Solution Domain Analysis is not sufficient*

There exist solution domain analysis approaches that are independent of software architecture design which provide systematic processes for identifying potentially reusable assets. As we have described before this activity is called *domain engineering* in the systematic reuse community. Unlike system engineering and problem domain engineering, solution domain analysis looks beyond a single system, a family of systems or the problem domain to identify the reusable assets within the solution domain itself. Although solution domain analysis provides the potential for modeling the whole domain that is necessary to derive the architecture, it is not sufficient to drive the architecture design process. This is due to two reasons. First, solution domain analysis is not defined for software architecture design per se, but rather for systematic reuse of assets for activities in for example software development. Since the area on which solution domain analysis is performed may be very wide, it may easily result in a domain model that is too large and includes abstractions that are not necessary for the corresponding software architecture construction. The large size of the domain model may hinder the search for the architectural abstractions. The second problem is that the solution domain may not be sufficiently cohesive and stable to provide a solid basis for architectural design. Concepts in the corresponding may not have reached a consensus yet and the area may still be under development. Obviously, one cannot expect to provide an architecture design solution that is better than the solution provided by the solution domain itself. A thorough solution domain analysis may in this case also not be sufficient to provide stable abstractions since the concepts in the solution domain themselves are fluctuating.

3.4.4 Pattern-driven Architecture Design

Christopher Alexander's idea on pattern languages for systematically designing buildings and communities in architecture [Alexander 79] has been adopted by the software community and led to the so-called software *design patterns* [Gamma et al. 95]. Similar to the patterns of Alexander, software design patterns aim to codify and make reusable a set of principles for designing quality software. The software design patterns are applied for the design phase, though, the software community has started to define and apply patterns for the other phases of the software development process. At the implementation phase patterns or idioms [Coplien 92] have been defined to map object-oriented design to object-oriented language constructs. Others have defined patterns for the analysis phase in which patterns are applied to derive analysis models [Fowler 96]. Patterns have also been applied at the architectural analysis phase of the software development process [PLOP][Buschmann et al. 96]. Architectural patterns are similar to the design patterns but focus on the gross-level structure of the system and its interactions. Sometimes architectural patterns are also called *architectural styles* [Shaw 95][Shaw & Garlan 96]. An architectural pattern is not the architecture itself, as it is often mistaken, but rather it is just an abstract representation at the architectural level [Abowd et al. 94] [Bass et al. 98].

Pattern-driven architecture design approaches derive the architectural abstractions from patterns. Figure 3.8 depicts the conceptual model for this approach.

The concept *Requirement Specification* represents a specification of a problem that may be solved using a pattern. The function *Search* represents the process for searching a suitable pattern for the given problem description and is supported by the concept *General Knowledge.*

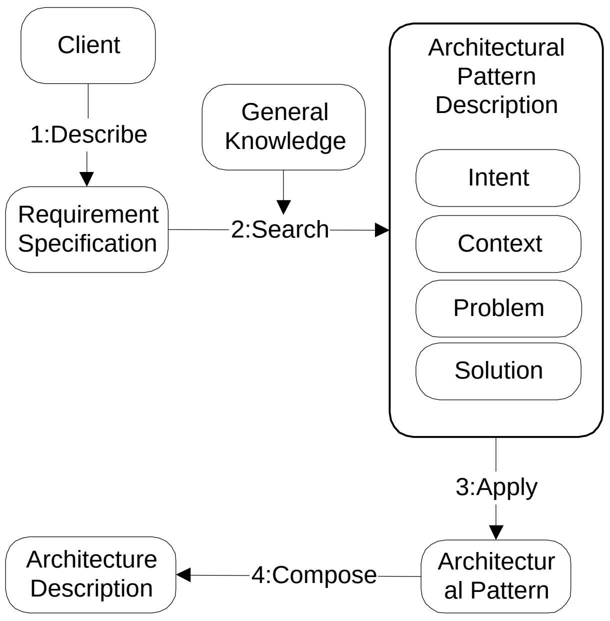


Figure 3.8 *Conceptual Model for a Pattern-Driven Architecture Design*

The concept *Architectural Pattern Description* represents a description of an architectural pattern. It consists mainly of four sub-concepts[24]: *Intent*, *Context*, *Problem*, and *Solution*. The concept *Intent* represents the rationale for applying the pattern. The concept *Context* represents the situation that gives rise to the problem. The concept *Problem* represents the recurring problem arising in the context. The concept *Solution* represents a solution to the problem in the form of an abstract description of the elements and their relations. For the identification of the pattern the intent of the available patterns is scanned. If the intent of a pattern is found relevant for the given problem then the context description (*Context*) is analyzed. If this also matches the context of the given problem, then the process follows with the function *3:Apply*. Thereby the sub-concept *Solution* is utilized to provide a solution to the problem. The concept *Architectural Pattern* represents the result of the function *3:Apply*. Finally, the function *4:Compose* represents the incorporation of the architecture pattern to the architecture description.

Problems

The pattern-driven architecture design approach is included as a sub-process in several architectural design approaches. Although architectural patterns are useful for building software architectures, the current approaches do not provide sufficient

[24] There are other sub-concepts but we consider these four sub-concepts as important for the identification of the architectural abstractions.

support for the selection of patterns, the application of these patterns and their composition to the architecture. We will describe these problems in the following:

- *Pattern base may not be sufficient for dealing with the wide range of architectural abstractions*

For a pattern-driven architecture design approach it is required that a sufficient base of patterns is available to support the design of software architectures. Currently, patterns have been catalogued in different publications such as [PLOP], [Buschmann et al. 96], [Shaw & Clements 97], [Gamma et al. 95], [Pree 95] and [Shaw 95]. Although, these catalogs provide practical vehicles for software architecture design, they do not and cannot cover all the problem areas for which architectures need to be developed. The reason for this is that architectures are composed of concepts representing abstractions from a particular domain and patterns define certain arrangements of these concepts and relations that are useful in solving recurring problems. Since there are numerous concepts and relations in the domain area, there are in principle also numerous architectural abstractions and accordingly numerous patterns. Consequently when utilizing a particular pattern catalogue, suitable patterns may be missing for a particular architecture design problem. In such cases it would be useful to provide means for generating new patterns for coping with novel but recurring problems.

- *Selecting patterns is merely based on the general knowledge and experience of the software engineer*

To ease the selection and manage and improve the understanding of patterns, patterns with common characteristics are usually classified into same groups. The classification criteria may differ per approach. For example, in [Shaw & Clements 97] and [Shaw 98] architectural patterns are classified according to the control and data interactions among architectural components. In [Buschmann et al. 96] patterns are classified into *problem categories*, grouping patterns addressing common problems. Sometimes, together with the categorization of the patterns a set of rules of thumb for choosing an architectural pattern is given as well. In [Shaw & Clements 97], for example, heuristic rules are given having the general form “If your problem has characteristic X then consider architectures with characteristic Y”. An example of such a heuristic rule is, “If your problem can be decomposed into sequential stages, consider batch sequential or pipeline architectures”. Despite of these classifications and heuristic rules it may happen that different alternative patterns are possible. Current approaches do not provide explicit support for prioritizing and balancing these alternative patterns. This is usually based on the experience and general knowledge of the software engineers. Therefore, this impedes the pattern-lookup process and as such the identification of the architectural abstractions.

- *Applying patterns is not straightforward and requires thorough analysis of the problem*

Once a pattern is selected the application of it is also not straightforward. A pattern is considered as a kind of template consisting of components and relations that must be matched with the concepts and concept relations identified in the problem domain. Examples of architectural patterns are *Pipes and Filters, Layering, Repositories, Interpreter,* and *Control* [Shaw & Garlan 96]. Assume for example, that for a given problem the architectural pattern *Pipes and Filters* is selected. In the *Pipes and Filters* architectural pattern the components are the *pipes* and the *filters* represent the connecting relations between the components. *Filters* are components that receive input streams, do some processing and provide some output. *Pipes* transmit the output stream of one filter to the input stream of another filter. Important questions in applying this pattern to the problem are: Which concepts should be represented as *Pipes*; which concepts should be represented as *Filters*; how should be the structuring of *Pipes* and *Filters* etc. Currently, there is no serious support for this matching process and pattern application is also based on the experiences and general knowledge of the software engineer.

- *Composing patterns is not well-supported*

For developing software architectures usually several individual patterns need to be composed. Patterns are generally not independent and reveal several relationships with each other. Specifying the patterns independently will not reflect these interdependencies. In [Buschmann et al. 96] patterns are collected and organized into problem categories providing a problem-oriented view in selecting and applying patterns. Systematic approaches with explicit guidelines for composing patterns, however, is missing[25]. Assume that after the problem analysis it appears that the patterns *Layering, Repositories,* and *Pipes and Filters* need to be composed in the architecture. In the *Layering* pattern the architecture components are represented through *layers* and the connectors are the protocols that determine the interactions between the layers. The *Repository* pattern consists of a shared data structure and a set of independent components that access the shared data structure. How should we compose these three patterns? Which should be the basic pattern? Why? What are the dependencies? Current pattern-driven approaches lack to provide satisfactory answers for these questions because patterns are specified independently.

[25] In architecture design, Alexander introduced the concept of *pattern language* that defines the structure and the mutual arrangement of the patterns as an integrated whole [Alexander 79].

3.5 Conclusion

In this chapter we have defined architecture as a set of abstractions and relations that form together a concept. Further, a meta-model that is an abstraction of software architecture design approaches is provided. We have used this model to analyze, compare and evaluate architecture design approaches. These approaches have been classified as *artifact-driven*, *use-case-driven*, *domain-driven* and *pattern-driven* architecture design approaches. The criterion for this classification is based on the adopted basis for the identification of the key abstractions of architectures. In the *artifact-driven* approaches the architectural abstractions are represented by groupings of artifacts that are elicited from the requirement specification. *Use-case driven* approaches derive the architectural abstractions from use case models that represents the system's intended functions. *Domain-driven* architecture design approaches derive the architectural abstractions from the domain models. *Pattern-driven* architecture design approaches attempt to develop the architecture by selecting architectural patterns from a pre-defined pattern catalogue. For each approach, we have described the corresponding problems and motivated why these sources are not optimal in identifying the architectural abstractions. We can abstract the problems basically as follows:

1. *Difficulties in Planning the Architectural Design Phase*

Planning the architecture design phase in the software development process is a dilemma[26]. In general architectures are identified before or after the analysis and design phases. Defining the architecture can be done more accurately after the analysis and design models have been determined because these impact the boundaries of the architecture. This may lead, however, to an unmanageable project because the architectural perspective in the software development process will be largely missing. On the other hand, planning the architecture design phase before the analysis and design phases may also be problematic since the architecture may not have optimal boundaries due to insufficient knowledge on the analysis and design models[27].

In artifact-driven architecture design approaches the architecture phase follows after the analysis and design phases and as such the project may become unmanageable.

[26] In [Aksit & Bergmans 92] this problem has been denoted as the *early decomposition* problem

[27] An analogy of this problem is writing an introduction to a book. To organize and manage the work on the different chapters it is required to provide a structure of the chapters in advance. However, the final structure of the introduction can be usually only defined after the chapters have been written and the complete information on the structure is available.

In the domain-driven architecture design approaches the architecture design phase follows a domain engineering phase in which first a domain model is defined from which consequently architectural abstractions are extracted. Hereby the architecture definition may be unmanageable if the domain model is too large. In the use-case driven architecture design approach the architecture definition phase is part of the analysis and design phase and the architecture is developed in an iterative way. This does not completely solve the dilemma since the iterating process is mainly controlled by the intuition of the software engineer.

2. *Client requirements are not a solid basis for architectural abstractions*

The client requirements on the software-intensive system that needs to be developed is different from the architectural perspective. The client requirements provide a problem perspective of the system whereas the architecture is aimed to provide a solution perspective that can be used to realize the system. Due to the large gap between the two perspectives the architectural abstractions may not be directly obvious from the client requirements. Moreover, the requirements themselves may be described inaccurately and may be either under-specified or over-specified. Therefore, sometimes it is also not preferable to adopt the client requirements.

This problem is apparent in all the approaches that we analyzed. In the artifact-driven and pattern-driven approaches the client requirements are directly used as a source for identifying the architectural abstractions. The use-case driven approach attempts to model the requirements also from a client perspective by utilizing use case models. In the domain-driven approaches, such as the domain specific software architecture design approach (DSSA), informal specifications are used to support the development of scenarios that are utilized to develop domain models.

3. *Leveraging the domain model is difficult*

The domain-driven and the use case approaches apply domain models for the construction of software architecture. Uncontrolled domain engineering may result in domain models that lack the right detail of abstraction to be of practical use. The one extreme of the problem is that the domain model is too large and includes redundant abstractions, the other extreme is that it is too small and misses the fundamental abstractions. Domain models may also include both redundant abstractions and still miss some other fundamental abstractions. It may be very difficult to leverage the detail of the domain model.

This problem is apparent in domain-driven and the use-case driven approaches. In the domain-driven approaches that derive domain models from problem domains, such as the DSSA approach, leveraging the domain model is difficult because it is based on scenarios that focus on the system from a problem perspective rather than a solution perspective. In the use-case driven architecture design approach, for example, leveraging the domain model and business model is difficult since it is performed before use-case modeling and it is actually not exactly known what is desired.

4. *Architectural abstractions have poor semantics*

A software architecture is composed of architectural components and architectural relations among them. Often architectural components are similar to groupings of artifacts, which are named as subsystems, packages etc. These constructs do not have sufficiently rich semantics to serve as architectural components. Architectural abstractions should be more than grouping mechanisms and the nature of the components and their relations, and the architectural properties, the behavior of the system should be described [Clements 96]. Because of the lack of semantics of architectural components it is very hard to understand the architectural perspective and make the transition to the subsequent analysis and design models.

5. *Composing architectural abstractions is weakly supported*

Architectural components interact, coordinate, cooperate and are composed with other architectural components. The architecture design approaches that we evaluated do not provide, however, explicit support for composing architectural abstractions.

Chapter 4

Architecture Synthesis Process

M.C.Escher's Waterfall © 2000 Cordon Art-Baarn-Holland. All rights reserved.

M.C. Escher - Waterfall

The illustration on the previous page shows one of Escher's famous impossible buildings. The basis of the illusion is the inclusion of the impossible triangle or tri-bar, developed by Roger Penrose and his father. The triangle is placed into the picture three times. As you look at each part of the construction in the print you cannot find any mistakes, but when the print is viewed as a whole you see the problem of water traveling up a flat plane, yet the water is falling and spinning a miller's wheel.

Software Architecture Design Analogy

Software architectures are usually developed from various individual architectural components, or sub-systems. Each individual architectural component may be perfectly designed and as such have a correct structure. However, the synthesis of the overall architecture may include serious flaws that may make the software architecture impossible to realize.

4.1 Introduction

"There are forces in nature called Love and Hate. The force of Love causes elements to be attracted to each other and to be built up into some particular form or person, and the force of Hate causes the decomposition of things."
- Empedocles

Research on software architecture design approaches is still in its progressing phase and several architecture design approaches have been introduced in the last years [Bass et al. 98], [Buschmann et al. 99], [Tracz & Coglianese 92], [Shaw 98]. However, a consensus on the appropriate software architecture design process is not established yet and current software architecture design approaches may have to cope with several problems.

First of all, planning the architecture design phase is intrinsically difficult due to its conflicting goals of providing a gross level structure of the system and at the same time directing the subsequent phases in the project. The first goal requires planning the architecture in later phases of the software development process when more information is available. The latter goal requires planning it as early as possible so that the project can be more easily managed.

Second, most software architecture design approaches derive the architectural abstractions in different ways and from different sources such as artifacts, use-cases, patterns and problem domains. These sources are basically focused on the client's-perspective[28] rather than on the architectural solution perspective of the system. The gap between the client perspective and the architectural design perspective is generally too large and the client may lack to specify the right detail of the problem, thereby either under-specifying or over-specifying the problem. This on its turn hinders the identification of the right architectural abstractions since the fundamental transparent abstractions may be missed or redundant abstractions may be elicited.

Third, generally the adopted sources are also not very useful to provide sufficiently rich semantics of the architectural components and fall short in providing guidelines for composing the architectural abstractions. In this case, architectural components are often equivalent to semantically poor groupings of artifacts and are composed using simple associations.

[28] We use the term client to denote any stakeholder who has interest in the application of a software architecture.

Finally, although solution domain analysis may be used and be effective in deriving the architectural abstractions and provide the necessary semantics, it may not suffice if it is not managed well. The problem is that the domain model may lack the right detail of abstraction to be of practical use for deriving architectural abstractions.

Current architecture design approaches have to cope with one or more of the above problems. In this chapter, a novel approach is proposed, which is termed *synthesis-based software architecture design* that aims to provide effective solutions to these problems. *Synthesis* is a well-known concept in traditional engineering disciplines and involves the construction of sub-solutions for distinct loosely coupled sub-problems and the integration of these sub-solutions into a complete solution. During the synthesis process design alternatives are searched and selected based on the existing solution domain knowledge.

In the synthesis-based software architecture design approach, the synthesis concept of traditional engineering disciplines is applied to the software architecture design process. Hereby, the requirements are first mapped to technical problems. For each problem the corresponding solution domain is identified and architectural abstractions are derived from the solution domain knowledge. The solution domain knowledge provides well-established concepts with rich semantics and as such form a stable basis for architecture development. The individual sub-solutions are combined in the overall software architecture.

In this chapter we will demonstrate the approach using a project on the design of an atomic transaction system architecture for a distributed car dealer information system[29].

The remainder of the chapter is organized as follows. In section 4.2, the synthesis concept is described and a model for software architecture synthesis is derived. In section 4.3, an example project on the design of a software architecture for atomic transactions for a distributed car dealer information system will be described. This example project will be used throughout the whole chapter. In section 4.4, the synthesis-based architecture design approach will be presented that will be illustrated for the example project. Finally, in section 4.5, we will present our discussion and conclusions.

[29] This work has been carried out as part of the INEDIS project that was a cooperative project between Siemens-Nixdorf and the TRESE group, Software Engineering, Dept. of Computer Science, University of Twente.

4.2 Synthesis

This section describes the concept of synthesis. *Synthesis* is a well-known concept in traditional engineering disciplines and is widely applied to solve design problems [Maher 90]. Software architecture design can be considered as a problem solving process in which the problem represents the requirement specification and the solution represents the software architecture design. In this section we apply the synthesis process to the software architecture design process. In section 4.2.1 we will explain the concept *synthesis* as it is described in traditional engineering disciplines. In section 4.2.2 we will apply the concept synthesis to software architecture design and gradually derive the steps for defining the software architecture synthesis model.

4.2.1 Synthesis in Traditional Engineering

Synthesis in engineering often means a process in which a problem specification is transformed to a solution by first decomposing the problem into loosely coupled sub-problems that are independently solved and integrated into an overall solution. In particular, the synthesis process includes an explicit phase for searching design alternatives in the corresponding solution domain and selecting these alternatives based on explicit quality criteria.

Synthesis consists generally of multiple steps or cycles. A synthesis cycle corresponds to a transition (transformation) from one *synthesis state* to another and can be formally defined as a tuple consisting of a problem specification state and a design state [Maimon & Braha 96]. The problem specification state defines the set of problems that still needs to be solved. The design state represents the tentative design solution that has been lastly synthesized. Initially, the design state is empty and the problem specification state includes the initial requirements. After each synthesis state transformation, a sub-problem is solved. In addition a new sub-problem may be added to the problem specification state.

Each transformation process involves an evaluation step whereby it is evaluated whether the design solutions so far (design state) are consistent with the initial requirements and any additional requirements identified during the synthesis.

A *synthesis-based design process* is defined as a finite sequence of synthesis states, resulting in a terminal state. A synthesis state is terminal in either of two cases: the specification part is satisfiable by the design part (there is a solution) or neither the design nor the specification can be modified. The first is a successful design the latter is an unsuccessful one.

The sub-solutions and overall solution has to meet a set of objective metrics, while satisfying a set of constraints. Constraints may be imposed within and among the sub-solutions. For a suitable synthesis it is required that the problem is understood well. This means that the problem is well-described and the quality criteria and constraints are known on beforehand. In practice, however, this is very difficult to meet and complete analysis is impossible in any but the simplest problems [Coyne et al. 90]. Therefore, synthesis can usually start before the problem is totally understood.

During the synthesis process a designer needs to consider the design space that contains the knowledge that is used to develop the design solution. For this, synthesis requires the ability to produce a set of alternative solutions and select an optimal or near optimal solution. The space of possible solutions, however, may be very large and it is not feasible to examine all possible solutions [Coyne et al. 90].

In [Maimon & Braha 96] it has been shown that the design synthesis is inherently an NP-Complete problem. To manage this inherent complexity, synthesis can be performed at different, higher abstraction levels in the design process. In the design of digital signal processing systems, for example, the following synthesis approaches with increasing abstraction levels are distinguished: circuit synthesis, logic synthesis, register-transfer synthesis, and system synthesis [Gajski et al. 92]. For large problems, the lower-level design synthesis approaches become intractable and time consuming due to the large number of entities and their relations that need to be considered. In the example of digital signal processing, circuit synthesis adopting the transistor as the basic abstraction, is unsuitable for current industrial problems that integrate millions of components. A higher level of abstraction reduces the number of entities that a designer has to consider which in turn reduces the complexity of the design of larger systems. In addition, higher level abstractions are closer to a designer’s way of thinking and as such increases the understandability, which on its turn facilitates to consider various alternatives more easily. The counterpart is that higher level abstractions consists of the fixed configuration of lower level abstractions thereby reducing the alternative configuration possibilities, that is, the set of alternatives is implicitly reduced. This is acceptable, though, since usually the total space of a synthesis from higher level abstractions is large enough to be of practical use.

4.2.2 Defining the Software Architecture Synthesis Model

Mapping Client Requirements to Technical Problems

Client requirements may lack to specify the right detail of the problem and either under-specify or over-specify the problem domain. Therefore, the gap between the requirements and the architectural design solution is generally too large. To solve this problem we propose to introduce a *technical problem analysis* phase that functions as an intermediary process between the requirements analysis and architectural design. Within this technical problem analysis phase, the delivered client requirements are thoroughly analyzed and mapped to technical problems that describe the problems more accurately. In this way, the gap between the requirements and the architectural design is largely reduced and, once the problems are clearly understood and specified independently of the initial requirements, a solid basis is provided to drive the architecture development.

Figure 4.1 illustrates the separation of the concept *Technical Problem* from the concept *Requirements*. The rounded rectangles represent the concepts; the directed arrows represent the functions between the concepts. The left side of the figure before the hollow arrow represents the approach that is adopted in several architecture design approaches. Hereby, the requirements are basically directly mapped to the (architectural) solution abstractions. The right side of the figure represents the introduction of the concept *Technical Problem* that has been separated from the concept *Requirements*. Hereby the concept *Requirements* is not directly mapped to the concept *Solution* but first it is analyzed and mapped to the concept *Technical Problem*. The concept *Technical Problem* describes the fundamental aspects that may not have been present in the original requirements. A clear understanding of the problem is part of the solution and as such this reduces the distance to the final solution.

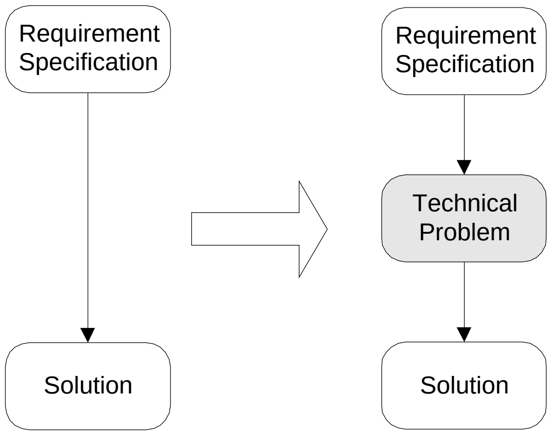


***Figure 4.1** The separation of the concept Technical Problem from the concept Requirements*

Deriving Architectural Abstractions from Solution Domain Models

We maintain that architectural abstractions should be best derived from the solution domain knowledge. The reason for this is threefold:

First, the solution domain knowledge includes well-established concepts that will not change abruptly. This is because solution domain knowledge is defined by a thorough analysis and research and is sufficiently stabilized through a consensus of experts in the corresponding community. A basic requirement for architectural components is that they should be stable and solution domain concepts provide this stability.

Second, the solution domain concepts are semantically rich, and define the properties, the relations with other concepts and their behavior. The required semantics for the architectural components may be derived from the solution domain concepts.

Third, solution domain concepts are related to each other and structured into taxonomies and part-whole relations. Further, the compatibility and composition relations between the various concepts are also well-defined. Existing architectural design approaches provide weak support for composing the architectural components as we have described in chapter 3. Solution domain knowledge provides the necessary information to support the composition of architectural components.

There exist domain analysis approaches that aim to provide solution domain models [Prieto-Diaz & Arrango 91] [Arrango 94] [Wartik & Prieto-Diaz 92]. We argue that these approaches should be integrated within the architecture design approaches to derive stable architectures.

Leveraging Solution Domain Models to the Identified Technical Problems

The solution domain analysis should be appropriately managed so that the domain model is optimally tuned to the architectural design phase. The right level of detail of the solution domain model can only be defined if both the client’s requirements and the corresponding solution domain are considered. On the one hand, the initial client requirements will likely fail to accurately define the overall-scope and the relevant abstractions because it does not provide a solution perspective of the problem. On the other hand, the solution domain itself may be very large and include abstractions that are not relevant for solving the corresponding problem. We maintain that the separated problem specifications from the client requirements provide a useful basis for leveraging the solution domain knowledge. This is because it is supposed to describe all the necessary fundamental aspects for solving the problem.

This requirement is illustrated in Figure 4.2, which is a refinement of Figure 4.1. The refinement here consists of the introduction of the concept *Solution Domain Knowledge*.

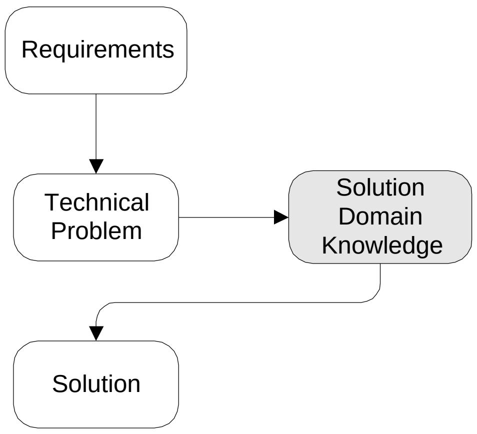


Figure 4.2 *Leveraging solution domain knowledge to the problems*

The arrow from the concept *Solution Domain Knowledge* to the concept *Solution* represents the previous requirement of deriving solution abstractions from solution domain knowledge. The arrow from the concept *Technical Problem* to the concept *Solution Domain Knowledge* represents the search and leveraging of the solution domain knowledge by the identified problems.

Defining Architecture Iteratively and Recursively

Planning the architectural design phase is intrinsically difficult due to the conflicting goals of its intended use. On the one hand it needs to represent the gross-level structure of the system and for this it is necessary to have a complete overview of the system including the analysis and design models in the later phases of the software development process. In addition, it is intended to be used to manage the project adequately and this requires defining the architecture as early as possible.

We require that architectures should be derived from solution domain knowledge that we proposed to leverage according to the identified problems as it has been illustrated in Figure 4.2.

To solve this dilemma we argue to adopt both an iterative and a recursive architecture design approach. Iteration means that the same steps of a process are repeated to correct what has already been done. Recursion means that the same steps of a process are repeated for a lower abstraction level. For architecture design approach iteration means that the process is repeated to correct the architectural abstractions because of newly acquired information. Recursion means that the same process is repeated to define the sub-architectural concepts. The process of iteration and recursion is visualized in Figure 4.3.

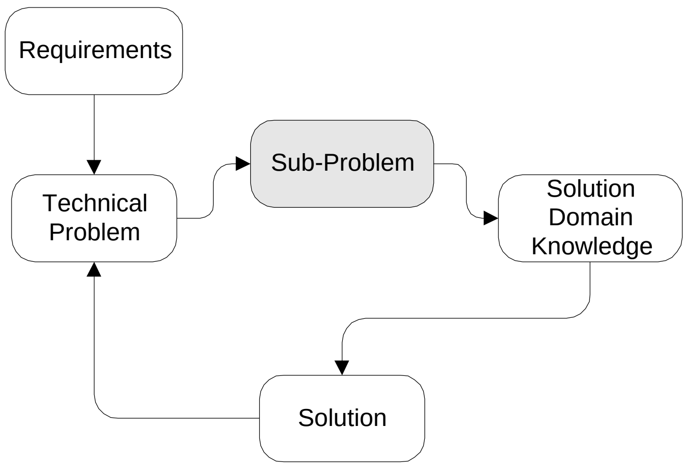


Figure 4.3 *Recursion and Iteration in providing Architectural Design Solution*

This figure is a refinement of Figure 4.2 and introduces the new concept *Sub-Problem*. The recursion process is basically defined by the decomposition of the problem into sub-problems whereby the suitable concepts are searched from the solution domain for each sub-problem individually. The iteration is represented by the arrow directed from the concept *Solution* to the concept *Problem*.

A Conceptual Model for Software Architecture Synthesis

Figure 4.4 represents a conceptual model for software architecture synthesis[30]. The model consists of two parts: *Solution Definition* and *Solution Control*. Each part consists of concepts and functions among concepts. The concepts are represented by rounded rectangles, the functions are represented by arrows. The part *Solution Definition* represents the identification and definition of solution abstractions. The part *Solution Control* represents the quantification, measurement, optimization and refinement of the selected solution abstractions. Note that this model represents a conceptual view of the software architecture synthesis process and does not enforce specific control flows between that implement various processes. In the following we will explain the concepts and functions of both parts of the model.

[30] Note that this model conforms to the CPC model described in chapter 2.

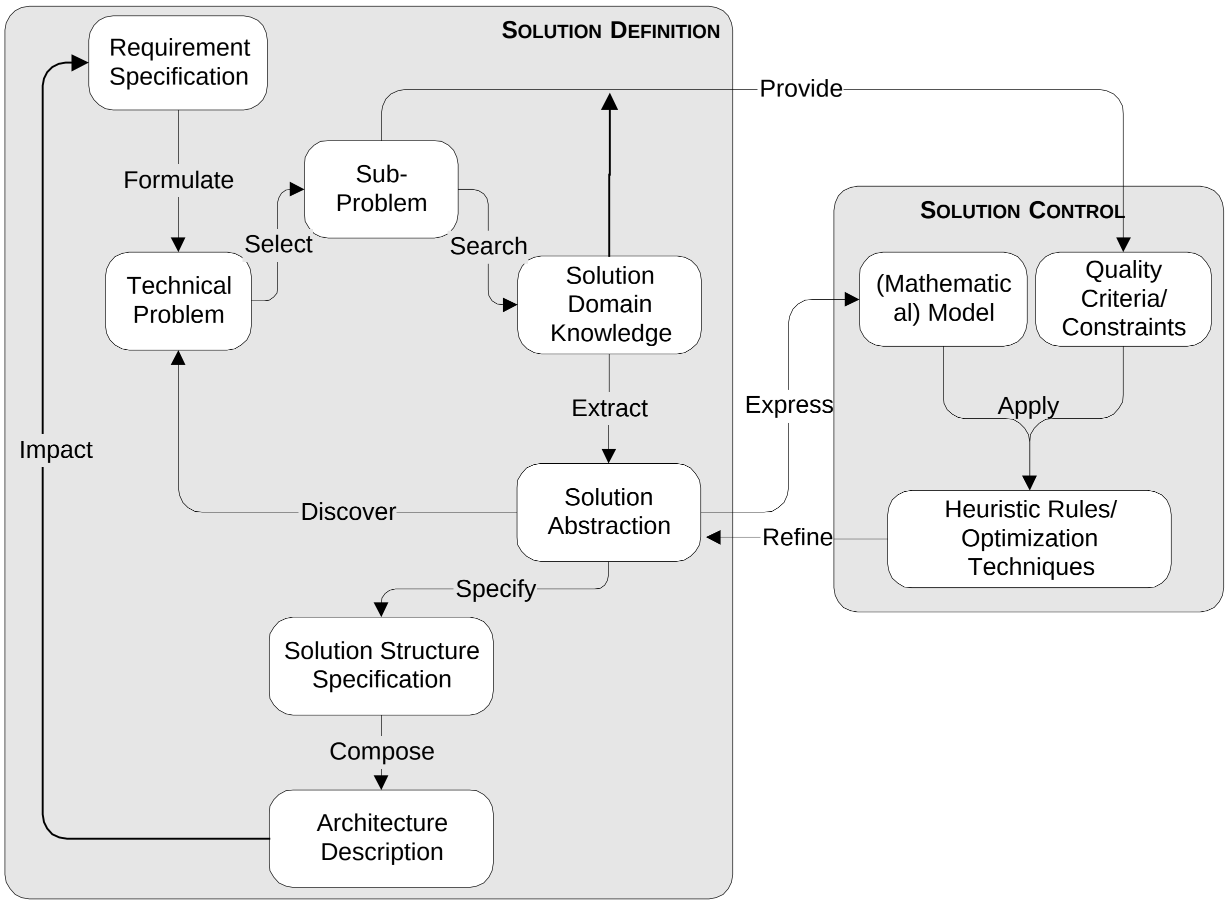


Figure 4.4 *Conceptual model for Architecture Synthesis*

Solution Definition

The concept *Requirement Specification* represents the requirements of the stakeholders who are interested in the development of a software architecture.

The concept *Technical Problem* represents the problem specification that is actually to be solved. The model thus separates the concepts *Requirement Specification* and *Technical Problem*.

The function *Formulate* defines the process for searching and representing the problems that need to be solved for the architecture development.

The concept *Sub-Problem* represents a sub-problem of the identified problem.

The function *Select* represents the process for selecting the corresponding sub-problem from the problem.

The concept *Solution Domain Knowledge* represents the solution domain knowledge that is needed for solving the sub-problem.

The function *Search* represents the process for searching the solution domain knowledge for a given problem.

The concept *Solution Abstraction* represents the extracted solution from the solution domain knowledge.

The function *Extract* represents the process for extracting the solution abstractions from the solution domain knowledge.

The concept *Solution Structure Specification* represents the specification of the extracted solution abstraction.

The function *Specify* represents the process for specifying the solution abstraction.

The concept *Architecture Description* represents the architecture description so far.

The function *Compose* represents the refinement of the overall-architecture description with the concept *Solution Structure Specification*.

The function *Discover* represents the process of discovering new sub-problems when new solution abstractions are extracted from the solution domain knowledge.

The function *Impact* represents the process of refining the requirement specification from the results of the architecture specification.

Solution Control

The part *Solution Control* has conceptual relations with the part *Solution Definition* through the functions *Provide, Express* and *Refine*.

The function *Provide* represents the process for providing the quality criteria and constraints that are imposed on the solution. The concept *Quality Criteria/Constraints* represents these criteria and constraints of the (sub-) problem. These are derived from the technical problems and / or the solution domain knowledge.

The function *Express* represents a formalization of the solution abstraction for evaluation purposes. Typical formalizations may be the quantification into mathematical models.

The function *Apply* represents the process for measurement of the expressed solution abstraction using the provided quality criteria / constraints.

The concept *Heuristic Rules / Optimization Techniques* represents the optimization of the formalizations of the solution abstractions. It can be based on mathematical optimization techniques or heuristic rules.

The function *Refine* represents the process of refining the solution abstraction according to the results of the optimization techniques.

4.3 Example Project: Transaction Software Architecture Design

The Integrated New European Dealer Information System project (INEDIS) has been carried out as a collaborative project between the TRESE group of the University of Twente and Siemens-Nixdorf, The Netherlands. The project dealt with the development of a distributed car dealer information system in which different car dealers are connected through a network. The system needs to provide automated support for processes such as workshop processing, order processing, stock management, *n*ew and used car management, and financial accounting. The car dealer system should execute the provided tasks consistently and effectively. The meaning of consistency depends on any a priori constraints, which must be guaranteed that they will never be violated. For example, two clients may not reserve the same car at the same time.

There are two main factors that threaten the consistency of data in a distributed system: *concurrency* and *failures*. In case of concurrency the executions of programs that access the same objects can interfere. When a failure occurs, one or more application programs may be interrupted in midstream. Since a program is written under the assumption that its effects are only correct if it would be executed in its entirety, an interrupted program may lead to inconsistencies as well. To achieve data consistency distributed systems should include provision for both concurrency and recovery from failures. The implementation of these concurrency and recovery mechanisms, however, should be transparent to the application program developers, since they will need only the primitives and don't want to be bothered with implementation details. *Atomic transactions*, or simply transactions, are a well-known and fundamental abstraction which provide the necessary concurrency control and recovery mechanisms for the application programs in a transparent way. Transactions relieve application programmers of the burden of considering the effects of concurrent access to objects or various kinds of failures during execution. Atomic transactions have proven to be useful for preserving the consistency in many applications like airline reservation systems, banking systems, office automation systems, database systems and operating systems.

The car dealer information system also required the use of atomic transactions. The system would be used in different countries and by different dealers each requiring dedicated transaction protocols. Therefore, a basic requirement of the system was to identify common patterns of transaction systems and likewise provide a stable architecture of atomic transactions that could be customized to the corresponding needs.

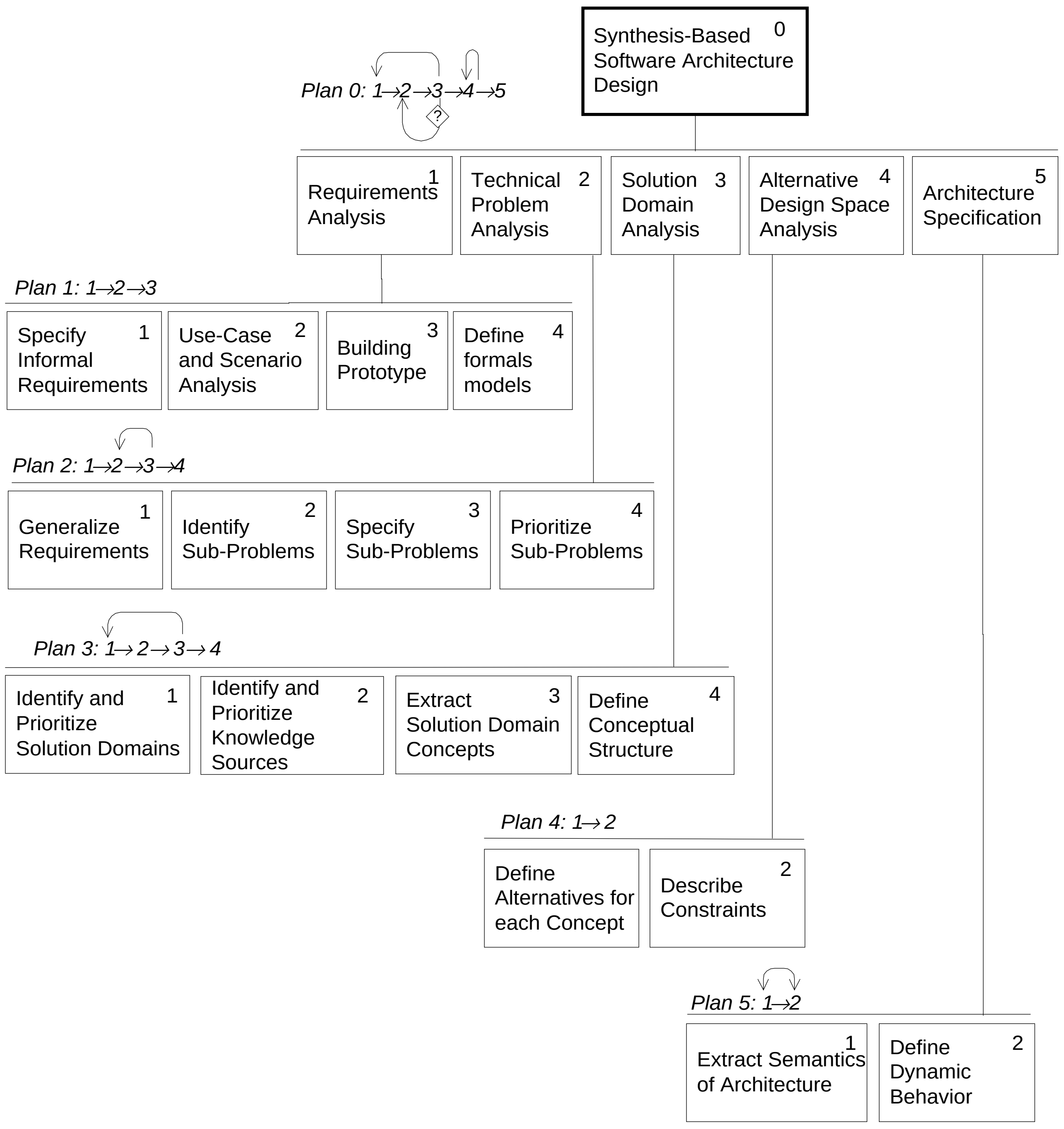
Next to the need for adaptability at initialization time the system required also adaptation at run-time. The system may be constituted of a large number of applications with various characteristics, operates in heterogeneous environments, and may incorporate different data formats. To achieve optimal behavior, this requires transactions with dynamic adaptation of transaction behavior, optimized with respect to the application and environmental conditions, and data formats. The adaptation policy, therefore, must be determined by the programmers, the operating system or the data objects. Further, reusability of the software is considered as an important requirement to reduce development and maintenance costs.

4.4 Synthesis-based Software Architecture Design

In this section the synthesis-based software architecture design process that implements the process of the Architecture Synthesis Model of Figure 4.4 will be described. This approach is illustrated in Figure 4.5.

The figure uses the graphical notation from Hierarchical Task Analysis (HTA) [Diaper 89b] in which activities are represented in hierarchical order. Each numbered box represents an activity that can be refined using a plan. Each plan represents a flow diagram describing the causal sequencing of the activities. The double-headed arrows represent interaction between two activities. The diamond with a question mark represents the validation of a step.

The following sections are organized around the basic process of the approach. Section 4.4.1 describes the *Requirements Analysis process*, section 4.4.2 the *Problem Analysis process,* section 4.4.3 the *Solution Domain Analysis process*, section 4.4.4 *Alternative Space Analysis process* and finally section 4.4.5 the *Architecture Specification process*.



Synthesis-Based Software Architecture Design 0
Plan 0: 1→2→3→4→5
?
Requirements Analysis 1
Technical Problem Analysis 2
Solution Domain Analysis 3
Alternative Design Space Analysis 4
Architecture Specification 5
Plan 1: 1→2→3
Specify Informal Requirements 1
Use-Case and Scenario Analysis 2
Building Prototype 3
Define formals models 4
Plan 2: 1→2→3→4
Generalize Requirements 1
Identify Sub-Problems 2
Specify Sub-Problems 3
Prioritize Sub-Problems 4
Plan 3: 1→2→3→4
Identify and Prioritize Solution Domains 1
Identify and Prioritize Knowledge Sources 2
Extract Solution Domain Concepts 3
Define Conceptual Structure 4
Plan 4: 1→2
Define Alternatives for each Concept
Describe Constraints 2
Plan 5: 1→2
Extract Semantics of Architecture 1
Define Dynamic Behavior 2

Figure 4.5 *Synthesis-based Software Architecture Design Approach*

4.4.1 Requirements Analysis

The architecture design is initiated with the requirements analysis phase in which the basic goal is to understand the stakeholder requirements. Stakeholders may be managers, software developers, maintainers, end-users, customers etc. [Prieto-Diaz & Arrango 91]. In the synthesis-based approach the well-known requirement analysis techniques such as textual requirement specifications, use-cases [Jacobson et al. 99] and scenarios [Kruchten 95], constructing prototypes and defining finite state machine modeling are used. Informal requirement specifications serve as a first basis for the requirements analysis process and is generally defined by interacting with the clients. Use cases provide a more precise and broader perspective of the requirements by specifying the external behavior of the system from different user perspectives. Scenarios are instances of use cases and define the dynamic view and the possible evolution of the system. Prototypes are used to define the possible user interfaces and may further help to clarify the desired behavior of the system. Finally, for safety-critical systems rigorous approaches such as state transition diagrams or formal specification languages may be used.

These techniques have been applied in different approaches and have shown to be useful in supporting the analysis and understanding of the client requirements. We will not elaborate on them in this thesis and refer for detailed information to the corresponding publications [Thayer et al. 97] [Sommerville & Sawyer 97] [Loucopoulos & Karakostas 95].

Example

At the start of the project the initial requirement specification was given by the client [Ahsmann & Bergmans 95]. We interviewed developers and managers of the project to extract the basic requirements for the INEDIS system [Tekinerdogan 95a]. We further analyzed the project literature, which included user's guide, manuals, design and implementation documentation and case studies. In addition we experimented with the existing NEDIS system in a real environment and identified the basic requirements for the further releases. Thereby, we were supported by the developer and maintainers of the system. Next to the overall requirements of the INEDIS system we focused on the requirements that were specific for atomic transactions [Tekinerdogan 95b][Tekinerdogan 96]. From this study we were able to set up the basic requirements that we expressed in use cases.

Figure 4.6 represents the use case model for transaction processing. It has one actor, Dealer, and four use cases, namely *initiate transaction, start transaction, abort transaction,* and *commit transaction*. The use case *initiate transaction* will be performed for describing and preparing a program to be used as a transaction. The use case *start transaction* will invoke the operations to access the transaction objects. Finally, the use cases *abort transaction* and *commit transaction* will describe the abort respectively the commit actions.

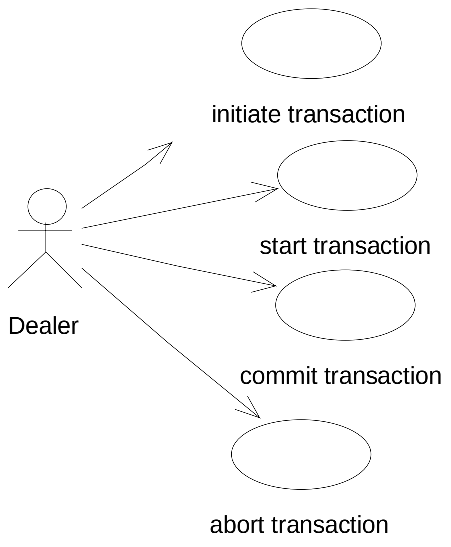


Figure 4.6 *A use case model for the transaction processing in the INEDIS system*

For a more detailed requirements analysis we refer to the project's requirements analysis documents [Tekinerdogan 95a][Tekinerdogan 95b][Tekinerdogan 96].

4.4.2 Technical Problem Analysis

The requirements analysis process provides an understanding of the client perspective of the software system. As it is described in Figure 4.5, the next step involves the technical problem analysis process in which client requirements are mapped to technical problems. This is to say that the architecture design process is to be considered as a problem solving process in which the solution represents an architecture design. The problem analysis process consists of the following steps:

1. *Generalize the Requirements:* whereby the requirements are abstracted and generalized.
2. *Identify the Sub-Problems*: whereby technical problems are identified from the generalized requirements.
3. *Specify the Sub-Problems*: whereby the overall technical problem is decomposed into sub-problems.
4. *Prioritize the Sub-Problems*: whereby the identified technical problems are prioritized before they are processed.

Let us explain these processes in more detail now.

Generalize the requirements

Discovering the problems from a requirement specification is not a straightforward task. The reason for this is that the clients may not be able to accurately describe the initial state and the desired goals of the system. The client requirements may be specific and provide only specific wordings of a more general problem. Therefore, to provide the broader view and identify the right problems we abstract and generalize from the requirement specification and try to solve the problem at that level[31]. Often, this abstraction and generalization process allows to define the client's wishes in entirely different terms and therefore may suggest and help to discover problems that were not thought of in the initial requirements.

Identify sub-problems

Once the requirement specification has been put into a more general and broader form, we derive the technical problem that consists usually of several sub-problems. At this phase, architecture design is considered as a problem solving process. Problem solving is defined as the operation of a process by which the transformation from the initial state to the goal is achieved [Newell & Simon 76]. We need thus first to discover and describe the problem. For this, in the generalized requirement specification we look for the important aspects that needs to be considered in the software architecture design [Tekinerdogan & Aksit 99]. These aspects are identified by considering the terms in the generalized requirements specification, the general knowledge of the software architect and the interaction with the clients. This process is supported by the results of the requirements analysis phase and utilizes the provided use-case models, scenarios, prototypes and formal requirements models.

Specify sub-problems

The identification of a sub-problem goes in parallel with its specification. The major distinction between the identification and the specification of a problem is that the first activity focuses on the process for finding the relevant problems, whereas the second activity is concerned with its accurate formalization. A problem is defined as the distance between the initial state and the goal. Thereby, the specification of the technical problems consists of describing its name, its initial state and its goal.

[31] In mathematics, solving a concrete problem by first solving a more general problem is termed as the *Inventor's Paradox* [Polya 57] [Lieberherr 96]. The paradox refers to the fact that a general problem has paradoxically a simpler solution than the concrete problem.

Prioritize sub-problems

After the decomposition of the problem into several sub-problems the process for solving each of the sub-problems can be started. The selection and ordering in which the sub-problems are solved, though, may have an impact on the final solution. Therefore, it is necessary to prioritize and order the sub-problems and process the sub-problems according to the priority degrees. The prioritization of the sub-problems may be defined by the client or the solution domain itself. The latter may be the case if a sub-problem can only be solved after a solution for another sub-problem has been defined.

Example

We generalized the INEDIS requirement specification [Ahsmann & Bergmans 95] and mapped these to the technical problems. For example, we generalized the requirements for the various scheduling techniques. In the original requirement specification and the interview with the stakeholders we identified that only two concurrency control approaches were used, namely optimistic and aggressive locking. Attempts were made to adapt between these two concurrency control mechanisms. After our discussion with the stakeholders [Tekinerdogan 95b] it followed that the system needed also other types of concurrency control protocols and the run-time adaptation had to be defined for these as well.

- *P0*
  Name: Provide adaptable architecture of atomic transactions
  Initial State: This is the overall problem. Initially no transaction architecture design was available.
  Goal: Identify the fundamental abstractions of transactions and design the atomic transaction software architecture that can be reused for different dealers and includes dynamic adaptation mechanisms for the different transaction protocols.

In parallel with our generalization of the requirements we were able to define the different sub-problems, which are listed in the following:

- *P1*
  Name: Provide transparent concurrency control.
  Initial State: Limited concurrency control techniques.
  Goal: Determine the set of concurrency control techniques that are required and provide this in a reusable form.

- *P2*
  Name: Provide transparent recovery techniques.
  Initial State: Limited recovery techniques based on simple data types.
  Goal: Determine the set of recovery techniques that can be used for various kinds of data types and provide this in a reusable form.
- *P3*
  Name: Provide transparent transaction management techniques.
  Initial State: Transaction management is primitive and is based on flat transactions only. The Start, Commit and Abort protocols are fixed.
  Goal: Provide various transaction management techniques that can be applied for advanced transactions such as long transactions and nested transactions. Provide the various start, commit and abort protocols in a reusable format.
- *P4*
  Name: Provide adaptable transaction protocols based on transaction, system and data criteria.
  Initial State: Selection of transaction protocols such as transaction management, concurrency control and recovery protocol is fixed.
  Goal: Provide the means to adapt the transaction protocols both on compile-time and run-time. Adaptation mechanism should be determined by programmers, operating system or the data object characteristics.

After interactive discussions with the stakeholders the above sub-problems have been prioritized in the given order, thus, P1, P2, P3 and P4. Figure 4.7 represents the *problem structure diagram*. In the problem structure diagram the nodes represent the technical problems and the lines the nesting relations. The nodes are numbered according to their nesting level. The problem structure diagram helps to sharpen and improve the understanding of the problem and can be used to reach a consensus with the client on the addressed problems. Note that the problem structure diagram in Figure 4.7 includes also the sub-problems of the problems that we described above. These sub-problems have been only identified during the refinement process after the solution domain analysis process. We will explain these in later sections. From this it follows that the problem structure diagram is not static but probably changes during the architecture design process because the problem may not be analyzed from a complete isolation from the solution domain [Cross 89].

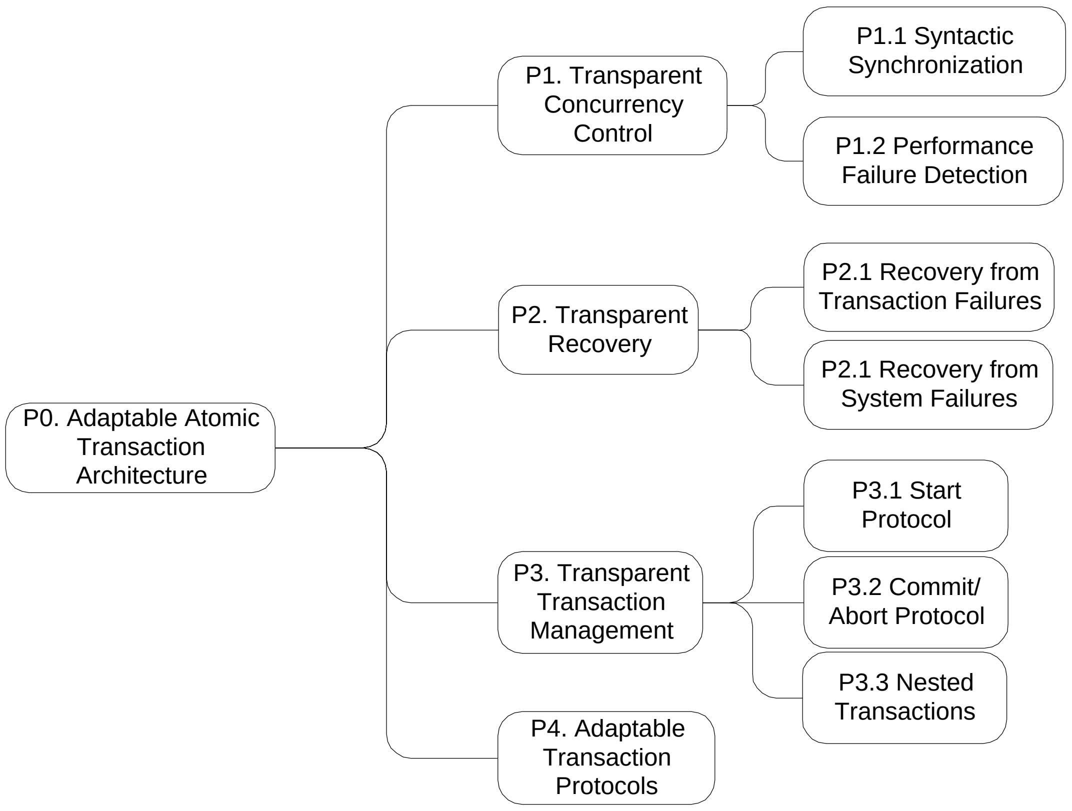

P0. Adaptable Atomic Transaction Architecture
P1. Transparent Concurrency Control
P1.1 Syntactic Synchronization
P1.2 Performance Failure Detection
P2. Transparent Recovery
P2.1 Recovery from Transaction Failures
P2.1 Recovery from System Failures
P3. Transparent Transaction Management
P3.1 Start Protocol
P3.2 Commit/ Abort Protocol
P3.3 Nested Transactions
P4. Adaptable Transaction Protocols

Figure 4.7 *Problem structure diagram for the example project*

4.4.3 Solution Domain Analysis

The Solution Domain Analysis process aims to provide a solution domain model that will be utilized to extract the architecture design solution. It consists of the following activities:

1. *Identify and prioritize the solution domains for each sub-problem*
2. *Identify and prioritize knowledge sources for each solution domain.*
3. *Extract solution domain concepts from solution domain knowledge.*
4. *Structure the solution domain concepts.*
5. *Refine the solution domain concepts.*

In the following we will explain these sub-processes. In Figure 4.5, the first four activities are represented from plan 3.1 to plan 3.4. The refinement of the solution domain concepts is represented by a directed arrow from plan 0.3 to plan 0.1.

To understand the relations between these activities Figure 4.8 represents a conceptual model for illustrating the relations between the concepts *Technical Problem, Sub-Problem, Solution Domain,* and *Solution Domain Concept.*

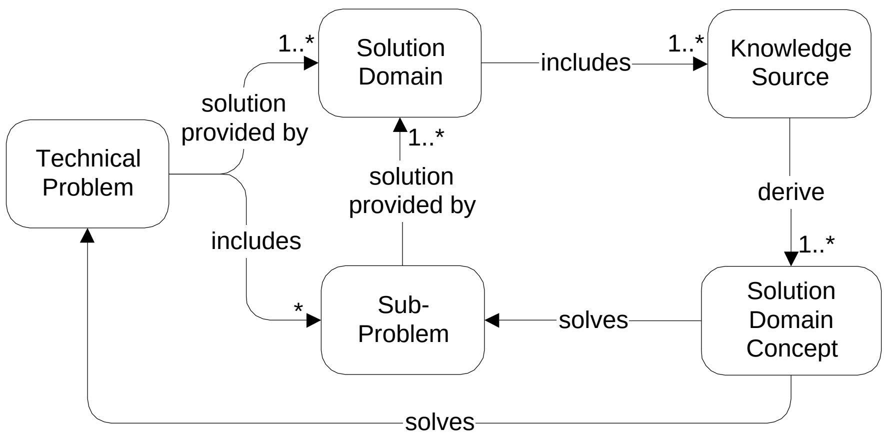


Figure 4.8 *The relations from Problem to Solution Domain Concept*

Hereby, the rounded rectangles represent the concepts and the directed arrows represent the associations between these concepts. From the figure it follows that for each *Technical Problem* a solution is provided by one or more *Solution Domains.* The concept *Problem* includes zero or more *Sub-Problems.* Each *Solution Domain* includes 1 or more *Knowledge Sources* from which 1 or more *Solution Domain Concepts* may be derived that solves the concepts *Problem* and *Sub-Problem.*

Identify and Prioritize the Solution Domains

For the overall problem and each sub-problem we search for the solution domains that provide the solution abstractions to solve the technical problem. The solution domains for the overall problem are more general than the solution domains for the sub-problems. In addition, each sub-problem may be recursively structured into sub-problems requiring more concrete solution domains on their turn.

An obstacle in the search for solution domains may be the possibly large space of solution domains leading to a time-consuming search process. To support this process, we look for categorizations of the solution domain knowledge into smaller sub-domains. There are different categorization possibilities [Glass & Vessey 95]. In library science, for example, the categories are represented by *facets* that are groupings of related terms that have been derived from a sample of selected titles [Rubin 98]. In [Aksit 00], the solution domain knowledge is categorized into *application*, *mathematical* and *computer science* domain knowledge. The application domain knowledge refers to the solution domain knowledge that defines the nature of the application, such as reservation applications, banking applications, control systems etc. Mathematical solution domain knowledge refers to mathematical knowledge such as logic, quantification and calculation techniques, optimization techniques, etc. Computer science domain refers to knowledge on the computer science solution abstractions, such as programming languages, operating systems, databases, analysis and design methods etc. This type of knowledge has been recently compiled in the so-called Software Engineering Body of Knowledge (SWEBOK) [Bourque et al. 99]. Notice that our approach does not favor a particular categorization of the solution domain knowledge and likewise other classifications besides of the above two approaches may be equally used.

If the solution domains have been adequately organized one may still encounter several problems and the solution domain analysis may not always warrant a feasible solution domain model. This is especially the case if the solution domains are not existing or the concepts in the solution domain are not fully explored yet and/or compiled in a reusable format. Figure 4.9 shows the flow diagram for the feasibility study on solution domain analysis. Hereby, the diamonds represent decisions, the rectangles the processes and the rounded rectangle the termination of the flow process.

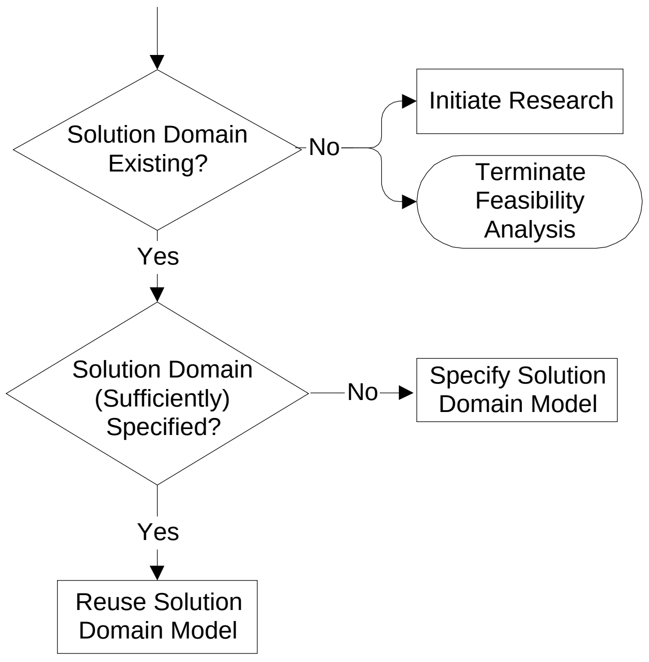

Solution Domain Existing?
No
Initiate Research
Terminate Feasibility Analysis
Yes
Solution Domain (Sufficiently) Specified?
No
Specify Solution Domain Model
Yes
Reuse Solution Domain Model

Figure 4.9 *Flow diagram for feasibility study on solution domain analysis*

If the solution domain knowledge is not existing, one can either terminate the feasibility analysis process or initiate a scientific research to explore and formalize the concepts of the required solution domain. The first case leads to the conclusion that the problem is actually not (completely) solvable due to lack of knowledge. The latter case is the more long-term and difficult option and falls outside the project scope.

If a suitable solution domain is existing and sufficiently specified, it can be (re)used to extract the necessary knowledge and apply this for the architecture development. It may also happen that the solution domain concepts are well-known but not formalized [Shaw & Garlan 96]. In that case it is necessary to specify the solution domain.

Identify and Prioritize Knowledge Sources

Each identified solution domain may cover a wide range of solution domain knowledge sources. These knowledge sources may not all be suitable and vary in quality. For distinguishing and validating the solution domain knowledge sources we basically consider the quality factors of *objectivity* and *relevancy*. The objectivity quality factor refers to the solution domain knowledge sources itself, and defines the general acceptance of the knowledge source. Solution domain knowledge that is based on a consensus on a community of experts has a higher objectivity degree than solution domain knowledge that is just under development. The relevancy factor refers to the relevancy of the solution domain knowledge for solving the identified technical problem.

The relevancy of the solution domain knowledge is different from the objectivity quality. A solution domain knowledge entity may have a high degree of objective quality because it is very precisely defined and supported by a community of experts, though, it may not be relevant for solving the identified problem because it addresses different concerns. To be suitable for solving a problem it is required that the solution domain knowledge is both objective and relevant. Therefore, the identified solution domain knowledge is prioritized according to their objectivity and relevancy factors. This can be expressed in the empirical formula [Aksit 00]:

$$priority(s) = (objectivity(s), (relevance(s))$$

Hereby *priority, objectivity* and *relevance* represent functions that define the corresponding quality factors of the argument *s*, that stands for solution domain knowledge source. For solving the problem, first the solution domain knowledge with the higher priorities is utilized. The measure of the objectivity degree can be determined from general knowledge and experiences. The measure for the relevancy factor can be determined by considering whether the identified solution domain source matches the goal of the problem. Note, however, that this formula should not be interpreted too strictly and rather be considered as an intuitive and practical aid for prioritizing the identified solution domain knowledge sources rather.

Example

Let us now consider the identification and the prioritization of the solution domains for the given project example. For the overall problem, a solution is provided by the solution domain *Atomic Transactions*. **Table 4.1** provides the solution domains for every sub-problem.

SUB-PROBLEM	SOLUTION DOMAIN
P1	Transaction Management
P2	Concurrency Control
P3	Recovery
P4	Adaptability

Table 4.1 *The solution domains for the sub-problems*

The prioritization of these solution domains was defined as in the above order from the top to the bottom.

ID	KNOWLEDGE SOURCE	FORM
KS1	*Concurrency Control & Recovery in Database Systems [Bernstein et al. 87]*	Textbook
KS2	*Atomic Transactions [Lynch et al. 94]*	Textbook
KS3	*An Introduction to Database Systems [Date 90]*	Textbook
KS4	*Database Transaction Models for Advanced Applications [Elmagarmid 92]*	Textbook
KS5	*The design and implementation of a distributed transaction system based on atomic data types [Wu et al. 95]*	Journal paper
KS6	*Transaction processing: concepts and techniques [Gray & Reuter 93]*	Textbook
KS7	*Principles of Transaction Processing [Bernstein & Newcomer 97]*	Textbook
KS8	*Transactions and Consistency in Distributed Database Systems [Traiger et al. 82]*	Journal paper

Table 4.2 *A selected set of the identified knowledge sources for the overall solution domain*

For the overall problem and the corresponding solution domain of *Atomic Transactions*, we could find sufficient knowledge sources. Our identified solution domain knowledge sources consisted of managers, system developers, maintainers, documentation, literature on transactions and the existing NEDIS system. However, among these different knowledge sources we assigned higher priority values to the literature on atomic transaction systems. Table 4.2 provides the selected set of knowledge sources for the overall solution domain.

The table consists of three columns that are labeled as *ID*, *Knowledge Source* and *Form* that respectively represent the unique identifications of the knowledge sources, the title of the knowledge source and the representation format of the knowledge source. The table includes the knowledge sources that describe atomic transactions in a general way. Knowledge sources that deal with a specific aspect of transaction systems, for example such as deadlock detection mechanisms, have been temporarily omitted and are identified when the corresponding sub-problems are considered.

In the same manner we looked for knowledge sources for the individual sub-problems and we were able to identify many knowledge sources for the solution domains *Transaction Management*, *Concurrency Control* and *Recovery*. The solution domain *Adaptability* was more difficult to grasp than the other ones. For this we did a thorough analysis on the notion of adaptability and studied various possibly related publications such as control theory [Roxin 97][Foerster 79][Umplebey 90]. In addition we organized a workshop on Adaptability in Object-Oriented Software Development [Tekinerdogan & Aksit 97] [Aksit et al. 96].

As an example, Table 4.3 shows a selected set of the identified knowledge sources for the solution domain *Concurrency Control*.

ID	**KNOWLEDGE SOURCE**	**FORM**
KS1	*Concurrency Control in Advanced Database Applications [Barghouti & Kaiser 91]*	Journal paper
KS2	*Concurrency Control in Distributed Database Systems [Cellary et al. 89]*	Textbook
KS3	*The theory of Database Concurrency Control [Papadimitriou 86].*	Textbook
KS4	*Concurrency Control & Recovery in Database Systems [Bernstein et al. 87]*	Textbook
KS5	*Concurrency Control and Reliability in Distributed Systems [Bhargava 87]*	Journal paper
KS6	*Concurrency Control in Distributed Database Systems [Bernstein & Goodman 83]*	Textbook

Table 4.3 *A selected set of the identified knowledge sources for the solution domain CONCURRENCY CONTROl*

Note that the knowledge source KS4 has also been utilized for the overall solution domain. The reason for this is that this knowledge source is both sufficiently abstract to be suitable for the overall solution domain and provides detailed information on the solution domain *Concurrency Control.*

Extract Solution Domain Concepts from Solution Domain Knowledge

Once the solution domains have been identified and prioritized, the knowledge acquisition from the solution domain sources can be initiated. The solution domain knowledge may include a lot of knowledge that is covered by books, research papers, case studies, reference manuals, existing prototypes/systems etc. Due to the large size of the solution domain knowledge, the knowledge acquisition process can be a labor-intensive activity and as such a systematic approach for knowledge acquisition is required [Partridge & Hussain 95], [Gonzales & Dankel 93], [Wielinga et al. 92].

In our approach we basically distinguish between the *knowledge elicitation* and *concept formation* process. Knowledge elicitation focuses on extracting the knowledge and verifying the correctness and consistency of the extracted data. Hereby, the irrelevant data is disregarded and the relevant data is provided as input for the concept formation process. Knowledge elicitation techniques have been described in several publications and its role in the knowledge acquisition process is reasonably well-understood [Wielinga et al. 92], [Meyer & Booker 91], [Diaper 89a], [Firlej & Hellens 91].

The concept formation process utilizes and abstracts from the knowledge to form concepts[32]. In the literature, several concept formation techniques have been identified[33] [Parsons & Wand 97][Reich & Fenves 91][Lakoff 87]. One of the basic abstraction techniques in forming concepts is by identifying the variations and commonalities of extracted information from the knowledge sources [Stillings et al. 95][Howard 87]. Usually a concept is defined as a representation that describes the common properties of a set of instances and is identified through its name.

[32] Recall from chapter 3 that there are basically three views of concepts, including the classical view, the prototype view and the exemplar view. Concept forming through abstraction from instances is basically applied in the classical view and the prototype view [Lakoff 87].

[33] This process of concept abstraction is usually considered as a psychological activity that is often associated with the term ’experience’ [Stillings et al. 95]. Experts, i.e. persons with lots of experience, own a larger set of concepts and are better in forming concepts than persons who lack this experience.

Example

We analyzed and studied the identified solution domain knowledge according to the defined priorities and extracted the fundamental concepts. After considering the commonalities and variabilities of the extracted information from the solution domains we could extract the following solution domain concepts:

ATOMIC TRANSACTION SYSTEMS

An atomic transaction system is a well-known and fundamental abstraction which provide the necessary concurrency control and recovery mechanisms for the application programs. Transactions relieve application programmers of the burden of considering the effects of concurrent access to objects or various kinds of failures during execution. Transactions simplify the treatment of failures and concurrency and may thereby provide the application programmer location transparency, replication transparency, concurrency transparency and failure transparency. Informally atomic transactions are characterized by two properties: *serializability* and *recoverability* [Bernstein et al. 87]. Serializability means that the concurrent execution of a group of transactions is equivalent to some serial execution of the same set of transactions. Recoverability means that each execution appears to be all or nothing; either it executes successfully to completion or it has no effect on data shared with other transactions.

TRANSACTION

The concept *Transaction* represents a transaction block as defined by the programmer.

TRANSACTIONMANAGER

The concept *TransactionManager* provides mechanisms for initiating, starting and terminating the transaction. It keeps a list of the objects that are affected by the transaction. If a transaction reaches its final state successfully, then *TransactionManager* sends a commit message to the corresponding objects to terminate the transaction. Otherwise an abort message is sent to all the participating objects to undo the effects of the transaction. The *TransactionManager* concept includes knowledge about a variety of commit and abort protocols.

POLICYMANAGER

The concept *PolicyManager* determines the mechanisms for adapting transaction protocols. In most publications, the *PolicyManager* is included in the *TransactionManager*. We considered defining transaction policies as a different concern and therefore defined it as a separate concept.

SCHEDULER

The concept *Scheduler* is responsible for the concurrency control mechanism. It provides the concurrency control by restricting the order in which the operations are processed. Incoming operations may be accepted, rejected or put in a delay queue. Concurrency control may be based on syntactic ordering of the operations (e.g. read, write) or it may use semantic information of the transaction, such as information on the accessed data types. Traditional concurrency control techniques are locking, timestamp ordering and optimistic scheduling.

RECOVERYMANAGER

The concept *Recovery Manager* is responsible for the recovery in case of transaction aborts, system failures and / or media failures. Failures may have an effect on data objects and on transactions that read the data objects. Recovery of the data objects needs caching and undo / redo mechanisms. Recovery of the effected transactions requires scheduling for recovery so that failures are prevented.

DATAMANAGER

The concept *DataManager* controls the access to its object and keeps it consistent by applying concurrency control and recovery mechanisms. Further it may be responsible for the version management and the replication management of the data objects.

DATA OBJECT

The concept *Data Object* represents a data object that needs to be accessed in a consistent way. This means that the object must fulfill the consistency constraints set by the application.

Structure the Solution Domain Concepts

The identified solution domain concepts are structured using *generalization-specialization* relations and *part-whole* relations, respectively. In addition, also other structural *association* relations are used. Like the concepts themselves the structural relations between the concepts are also derived from the solution domains.

For the structuring and representation of concepts, so-called *concept graphs* are used. A *concept graph* is a graph which nodes represent *concepts* and the edges between the nodes represent *conceptual relations*. The notation of concept graphs is given in the following figure:

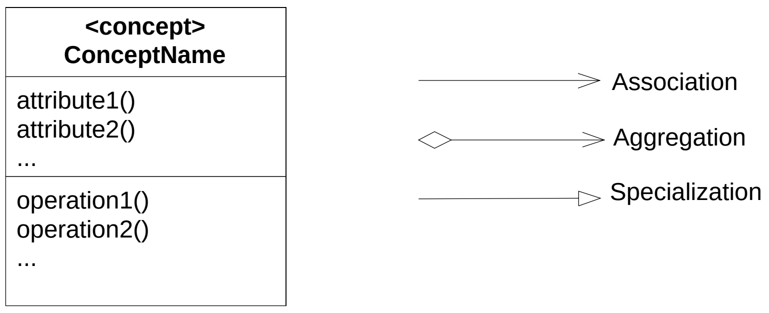


Figure 4.10 *Notation for concept graphs*

The notation for a concept is a stereotype of the class notation in the Unified Modeling Language [Booch et al. 99]. A stereotype represents a subclass of a modeling element with the same form but with a different intent. The stereotype for a concept Figure 4.10 is identified by the keyword <concept>[34].

Example

Figure 4.11 shows the structuring of the solution domain concepts in the top-level concept graph of transaction systems. The concept *Transaction Manager* has an association relation *manages* with the concept *Transaction*. This means that *Transaction Manager* is responsible for the atomic execution of *Transaction*. The association relation *manages* between concept *DataManager* and *Data Object* defines the maintenance of the consistency.

For keeping the *Data Object* consistent the *Datamanager* utilizes and coordinates the concepts *Scheduler* and *RecoveryManager* by means of the association relation *coordinates*. The concept *PolicyManager* coordinates the activities of the concepts *TransactionManager* and *DataManager* and defines the adaptation policy. Finally, the association relation *accesses* between *Transaction* and *Data Object* defines a read / update relation between these two.

[34] Note that a class may not be similar to a concept. Although both classes and concepts are generally formed through an abstraction process this does not imply that every abstraction is a concept. A concept is a well-defined and stable abstraction in a given domain. Although the notation that we use for representing concepts is similar to the notation of classes, one should be aware that concepts are at a different level than classes and should be treated as such.

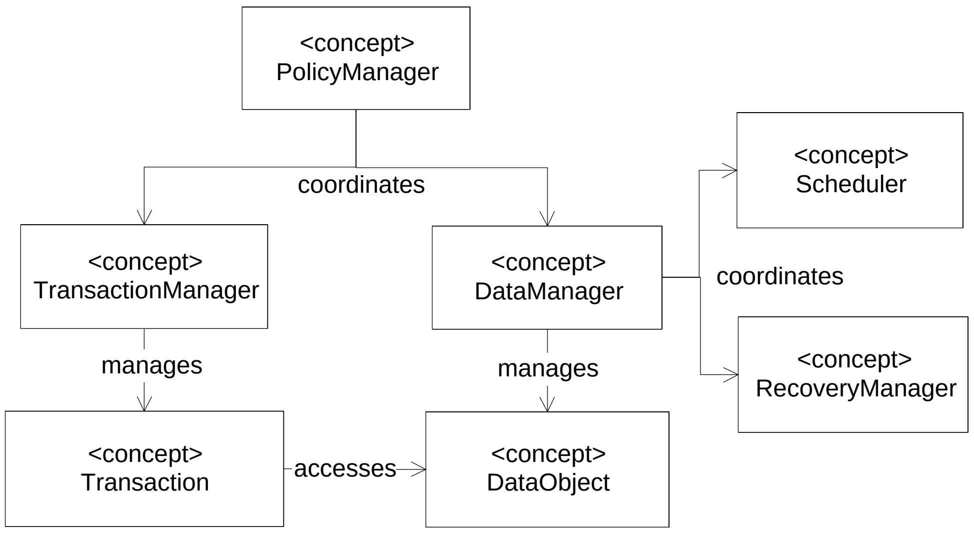


Figure 4.11 *The top-level concept graph of an atomic transaction system*

Refinement of Solution Domain Concepts

After identifying the top-level conceptual architecture we focus on each sub-problem and follow the same process. Recall that in Figure 4.5, this refinement process is represented by the arrow directed from plan 0.3 to plan 0.1. The refinement may be necessary if the architectural concepts have a complex structure themselves and this structure is of importance for the eventual system.

The ordering of the refinement process is determined by the ordering of the problems with respect to their previously determined priorities. Architectural concepts that represent problems with higher priorities are handled first. In the following we will refine the architectural concepts according to this ordering. The refinement requires executing the plans 0.1 to 0.3 for each selected concept. However, due to space limitations we will only describe these plans globally.

Example

In the following we will shortly describe the refinement for each concept of the atomic transaction architecture.

Refining the TransactionManager concept

To refine the concept *TransactionManager* we looked for the knowledge sources that specifically dealt with transaction management or included detailed information about this. We identified several publications for this purpose [Elmagarmid 92][Bernstein & Newcomer 97],[Moss 85][Jajodia & Kerschberg 97].

In parallel with the solution domain analysis process we tried to refine problem P3 for transparent transaction management, as it has been described in the problem structure diagram in Figure 4.7. This resulted in the definition of the sub-problems *P3.1 Start Protocol*, describing the need for defining a transaction start protocol, *P3.2 Commit/Abort Protocol*, describing the need for a commit/abort protocol and *P3.3 Nested Transactions*, describing the need for nested transactions. The specifications of these sub-problems were again defined in close interaction with the client. After comparison of the concepts in these knowledge sources we could extract the commonalities and derive the architecture for the concept *TransactionManager* as it is given in Figure 4.12.

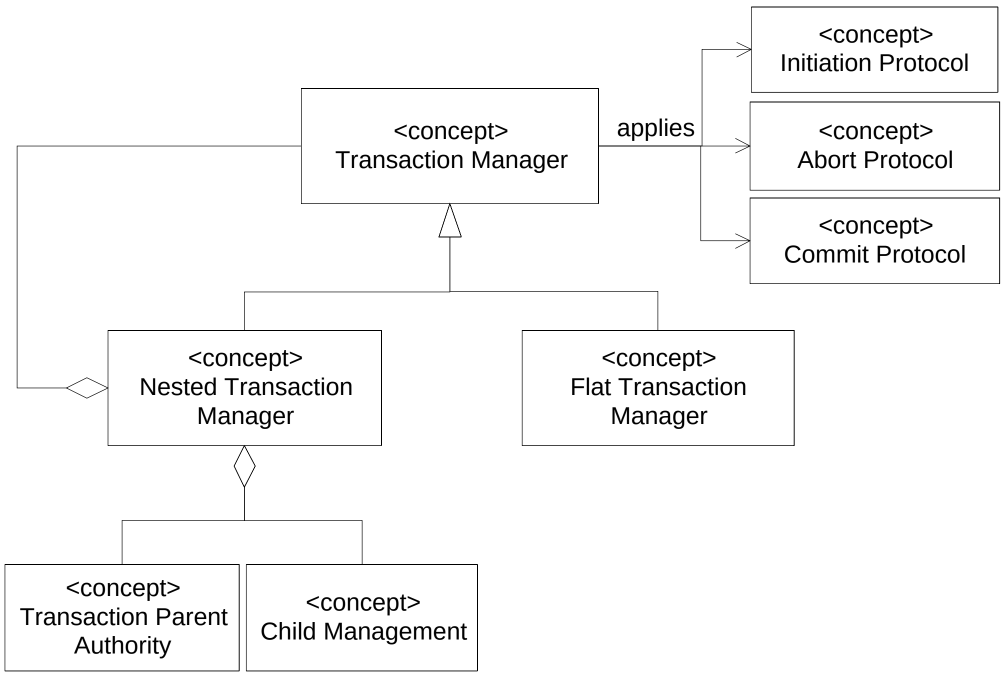

<concept> Initiation Protocol
<concept> Transaction Manager
applies
<concept> Abort Protocol
<concept> Commit Protocol
<concept> Nested Transaction Manager
<concept> Flat Transaction Manager
<concept> Transaction Parent Authority
<concept> Child Management

***Figure 4.12** Conceptual architecture of TransactionManager*

The concept *Transaction Manager* applies the concepts *Initiation Protocol, Abort Protocol* and *Commit Protocol* for starting and terminating a transaction. The *Initiation Protocol* represents the starting of the transaction and prepares the program to be executed. The *Abort Protocol* and *Commit Protocol* concepts refer to the protocols that terminate the transaction. The *Abort Protocol* will be executed if the transaction has failed and its effects on the data objects and the other transactions need to be restored. The *Commit Protocol* will be executed if the transaction protocol has succeeded and the results need to be made persistent.

Transactions may consist of other transactions as well. The composition of sub-transactions into one transaction is called a nested transaction [Moss 85]. Hereby the transactions are hierarchically ordered whereby a *parent transaction* includes several other sub-transactions. The advantages of nested transactions over flat transactions is that they provide internal parallelism of the sub-transactions and finer control over failures by limiting the effects of the failure to a sub-transaction. This is especially important for long and complex transactions that have, for example, higher failure risks. In Figure 4.12, the concept *Transaction Parent Authority* refers to the authority of the parent on the sub-transactions with respect to the commit protocols. In the literature, basically a distinction is made between *closed nested transactions* and *open nested transactions* [Elmagarmid 92]. In *closed nested transactions* the sub-transactions are not allowed to commit before the parent transaction commits, whereas in *open nested transactions* the sub-transactions can commit before the parent commits. The concept *Child Management* refers to the composition strategy of the sub-transactions into one complete transaction. Usually this is done at compile time, but several approaches have illustrated the practical use of dynamic composition and decomposition of sub-transactions [Pu et al. 88]

Refining the DataManager concept

Figure 4.13 shows the architecture for the concept *DataManager*. The basic knowledge sources that we adopted to identify the common abstractions of data management techniques are derived from several publications [Weihl 90][Wu et al. 95][Guerraoui 94].

The concept *Datamanager* coordinates the concepts *Scheduler* and *RecoveryManager*, which are respectively responsible for the scheduling of the incoming concurrent operations and the recovery in case of failures. In addition the concept *DataManager* uses the concepts *VersionManager* and *ReplicationManager* for respectively managing multiple versions of the data item and the replication of it at different locations. The version management and the replication management were not addressed as separate problems in the problem analysis phase. After interaction with the client it was decided to omit these two issues and only consider the concurrency control and recovery in the data management. If these were addressed as important problems then we would update the problem structure diagram and attempt to provide solutions for these problems in the later phases of the approach.

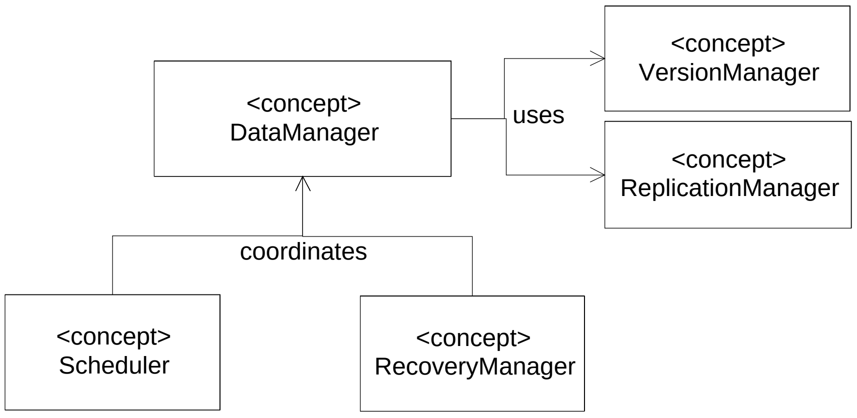


Figure 4.13 *Conceptual Architecture of DataManager*

Refining the Scheduler concept

Figure 4.14 represents the architecture of the concept *Scheduler*. The selected knowledge sources that we identified to extract this structure have been listed in Table 4.3 in the sub-section on identifying and prioritizing knowledge sources. In parallel with the solution domain analysis we refined the problem structure diagram for the concept *Scheduler* and added the sub-problems *P1.1 Syntactic Synchronization* and *P1.2 Performance Failure Detection*. These problems correspond to the solution domain concepts in the conceptual architecture of *Scheduler* that consists of three sub-concepts: *Synchronization Scheme, Synchronization Strategy* and *Performance Failure Detector.*

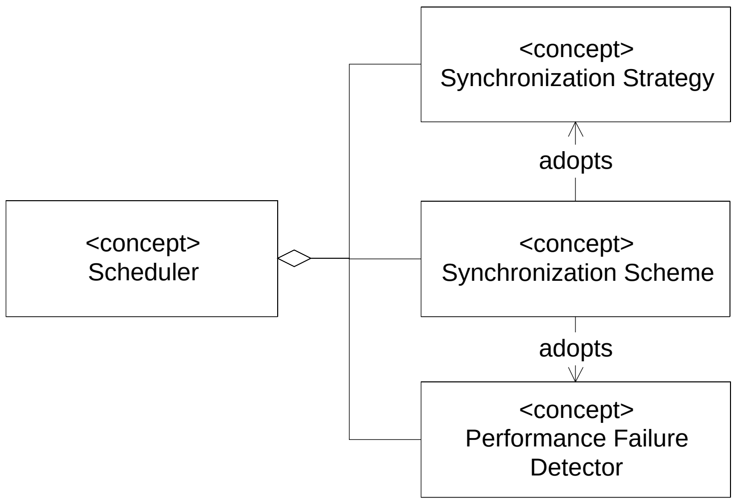


Figure 4.14 *Conceptual Architecture of Scheduler*

The concept *Synchronization Scheme* defines the synchronization approach by accepting, rejecting or delaying the incoming operations. It addresses the problem *P1.1 Syntactic Synchronization* in the problem structure diagram of Figure 4.7. The syntactic synchronization may be basically through locking, timestamp-ordering and optimistic concurrency control schemes. The concept *Synchronization Strategy* also addresses the problem *P1.1 Syntactic Synchronization* and refers to the adopted strategy in the applied concurrency control algorithm. Basically a distinction is made between conservative and aggressive schedulers. A conservative scheduler tends to delay operations whereas an aggressive scheduler avoids these delays and aborts the operation sooner. The concept *Performance Failure Detector* addresses the problem *P1.2 Performance Failure Detection* and concerns the detection of performance failures such as deadlocks that are side effects of the used concurrency control algorithms.

Refining the Recovery Manager Concept

The concept *RecoveyManager* is related to the problem *P2. Transparent Recovery* that is depicted in the problem structure diagram in Figure 4.7. It has been derived from the publications on recovery in transaction systems [Bernstein et al. 87][Bhargava et al. 86][Hadzilacos 88][Haerder & Reuter 83]. In parallel with refining the concept *RecoveyManager* we refined problem P2 and defined the sub-problems *P2.1 Recovery from Transaction Failures* and *P2.2 Recovery from System Failures*. The architecture for the concept *RecoveryManager* is given in Figure 4.15.

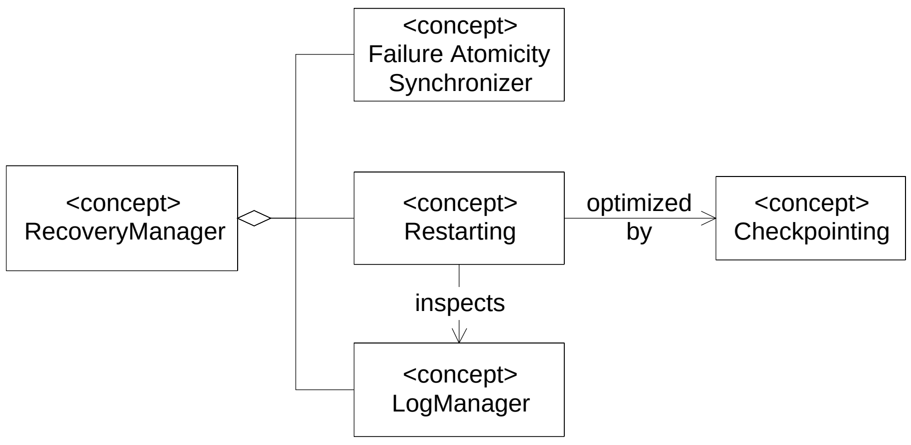


***Figure 4.15** Conceptual Architecture of RecoveryManager*

The concept *RecoveryManager* consists of four sub-concepts *Failure Atomicity Synchronizer*, *Restarting*, *LogManager* and *Checkpointing*. The effects of a transaction can be both on the accessed data objects and on other transactions that access the same data object. To undo the effects of failures on data objects the sub-concept *LogManager* is used for logging the data object. The sub-concept *Failure Atomicity Synchronizer* order transaction operations to provide the all-or-nothing property. The sub-concept *Restarting* is responsible for recovering from system failures and initializes the transaction to its last recoverable state by inspecting the logs of the *LogManager*. Thereby it uses algorithms for undoing the actions of aborted and active transactions and redoing the effects of transactions that have been committed before but not made persistent yet. The sub-concept *Checkpointing* represents the optimization of the restart process by making a snapshot or checkpoint of the basic events in the system that may be used by the protocols of the concept *Restarting*.

Refining the Policy Manager Concept

The concept *PolicyManager* is related to the technical problem *P4. Provide Adaptable Transaction Protocols*. The identified knowledge sources for the concept *PolicyManager* have been derived from several publications on control systems [Dorf & Bishop 98][Shinners 98] and performance modeling [Kumar 96][Atkins & Coady 92][Highleyman 89][Agrawal 87][Carey 84]. The *PolicyManager* evaluates a number of performance metrics and selects the preferred transaction protocols with respect to these parameters. Examples of performance metrics are the following:

1. *Transaction throughput rate,* which is the number of transactions completed per second.
2. *Response time,* which is the measure of the time difference between a transaction initiation and a successful termination of the transaction.
3. *Blocking ratio,* which is the average number that a transaction has to block per commit.
4. *Restart ratio,* which is the average number that a transaction has to restart per commit.

The conceptual architecture of *PolicyManager* is given in Figure 4.16. It consists of the sub-concepts *Sensor, Comparator, Decider* and *Actuator*.

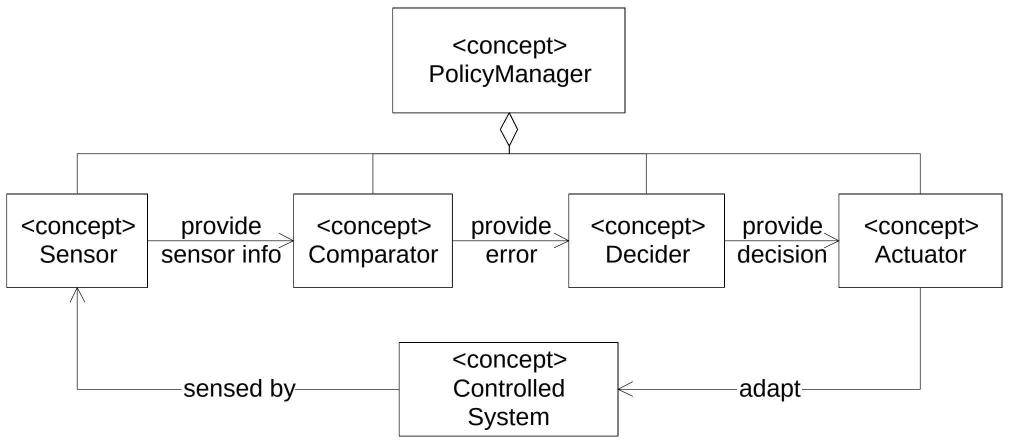


Figure 4.16 *Conceptual Architecture of PolicyManager*

The concept *Sensor* keeps track of the changes in the system state and the values of the performance parameters and provides this information to the sub-concept *Comparator* that compares this information according to the initialized criteria and goals that need to be met. For example, *Comparator* may have defined different threshold values for the *throughput* parameter and compares this with the perceived values of the *throughput* parameter and inform the sub-concept *Decider* about the difference. *Decider* will then select an adequate transaction protocol and inform the sub-concept *Actuator* about this decision. *Actuator* will actually adapt the system with the required transaction protocols. Note that this architecture may be implemented in various ways such as a very simple adaptation algorithm or an expert-system based selection. The flow of control between the different concepts can be implemented in different ways as well.

4.4.4 Alternative Design Space Analysis

We define the *alternative space* as the set of possible design solutions that can be derived from a given conceptual software architecture. The *Alternative Design Space analysis* aims to depict this space and consists of the sub-processes *Define the Alternatives for each Concept* and *Describe the Constraints*. Let us now explain these sub-processes in more detail.

Define the Alternatives for each Concept

In the synthesis-based design approach the various architecture design alternatives are largely dealt with by deriving architectural abstractions from well-established concepts in the solution domain that have been leveraged to the identified technical problems. Each architectural concept is an abstraction from a set of instantiations and during the analysis and design phases the architecture is realized by selecting particular instances of the architectural concepts. An instance of a concept is considered as an alternative of that concept. The total set of alternatives per concept may be too large and / or not relevant for solving the identified problems. Therefore, to define the boundaries of the architecture it is necessary to identify the relevant alternatives and omit the irrelevant ones.

Let us now consider the process of alternative selection. The alternatives of a given concept may be explicitly identified and published [Tekinerdogan 94]. In that case, selecting alternatives for a concept is rather straightforward and depends only on the solution domain analysis process. If the concepts have complex structures consisting of sub-concepts then an alternative is defined as a composition of instances of separate sub-concepts. The set of alternatives may be too large to provide a name for each of them individually. Nevertheless, we need to depict the total set of alternatives so that every one of them can be derived if necessary. We do this by identifying the alternatives of each sub-concept first and then considering the various compositions of these alternatives to provide the higher-level alternatives.

Example

Let us now consider the alternatives for the concepts in the top-level architecture. We depict the alternative space by providing a table in which the column headers represent the sub-concepts and each table entry represents an instance of the sub-concept in the column header. For example, Table 4.4 represents the alternative space for the concept *Scheduler.* It has 4 columns, the first one represents the numbering of alternatives and the second to the fourth columns represents the sub-concepts of the concept *Scheduler.*

	A. SYNCHRONIZATION SCHEME	**B. SYNCHRONIZATION STRATEGY**	**C. PERFORMANCE FAILURE DETECTOR**
1.	Two Phase Locking	Aggressive	Deadlock Detector
2.	Timestamp Ordering	Conservative	Infinite Blocking Detector
3.	Optimistic		Infinite Restart Detector
4.	Serial		Cyclic Restart Detector

Table 4.4. *Alternatives of the sub-concepts of* Scheduler

The sub-concept *Synchronization Decision* has four alternatives, namely, *Two-phase locking, Timestamp Ordering, Optimistic* and *Serial.* The sub-concept *Synchronization Strategy* has the alternatives *Aggressive* or *Conservative.* The alternatives of the sub-concept *Performance Failure Detector* detect *performance failures* that may consist of *deadlock, permanent blocking, cyclic restarting* and *infinite restarting. Deadlock* is defined as a state where two transactions are mutually waiting for each other to release data objects necessary for their completion. *Permanent blocking* occurs when a transaction waits indefinitely for a data object granting because of a steady stream of other transactions whose data access requests are always granted before. *Cyclic restarting* occurs when two or more transactions continually cause mutual abortion of each other. *Infinite restarting* occurs when a transaction is infinitely aborted because of a steady stream of other transactions whose operations are always granted before.

An alternative of the concept *Scheduler* is a composition of selections of the alternatives of the sub-concepts. For instance, an alternative that may be derived from Table 4.4 is the tuple *(Two Phase Locking, Conservative, Deadlock Detector*) which represents a scheduler that uses aggressive two phase locking protocol whereby a deadlock detection mechanism is used to remove the deadlocks that may occur.

Table 4.5 represents the alternative space for the concept *RecoveryManager*.

	A. LOGMANAGER	**B. FAILURE ATOMICITY SYNCHRONIZER**	**C. RESTARTING**	**D. CHECKPOINTING**
1.	Operation Logging	Recoverable	Undo / Redo	Commit-Consistent
2.	Deferred-Update	Cascadeless	No-Undo / Redo	Cache-Consistent
3.	Update-In-Place	Strict	Undo / No-Redo	Fuzzy
			No-undo / No-redo	

Table 4.5 *Alternatives of the sub-concepts of RecoveryManager*

The sub-concept *LogManager* consists of three alternatives of logging techniques. In the technique *Operation Logging*, the transaction’s operations that access a data object are logged. In case of aborts other operations are executed to undo the effects of the operations that were logged. Another logging technique is to make a copy of the state of the object, which is called *image logging*. Hereby, either the copy may be accessed or the original. The former is called *Deferred-Update Logging* because updates to the original data objects are deferred until commit time. The latter is called *Update-In-Place* logging whereby the copy of the data object is installed on abort and originals are left on commit. The sub-concept *Failure Atomicity Synchronizer* orders operation in three possible ways and provides either *recoverable, cascadeless* *aborts* or *strict* executions. Restarting can be performed as a combination of undo and redo protocols and as such there are four alternatives here. Finally, the sub-concept *Checkpointing* consists of the three alternatives *Commit-Consistent*, *Cache-Consistent* and *Fuzzy* checkpointing mechanisms.

An alternative of the concept *RecoveryManager* is the tuple *(Operation Logging, Strict, Undo-Redo, Commit-consistent)*, that represents a *RecoveryManager* which applies *Operation Logging*, *Strict* executions, adopts *Undo-Redo* algorithm in case of restarts and a *Commit-Consistent* checkpointing mechanism for optimizing the restart procedure.

Describe Constraints between Alternatives

An architecture consists of a set of concepts that are combined in a structure. An instantiation of an architecture is a composition of instantiations of concepts [Aksit et al. 99][Aksit et al. 98]. The instantiations of these various concepts may be combined in many different ways and likewise this may lead to a combinatorial explosion of possible solutions. Hereby, it is generally impossible to find an optimal solution under arbitrary constraints for an arbitrary set of concepts.

To manage the architecture design process and define the boundaries of the architecture it is important to adequately leverage the alternative space. Leveraging the alternative space means the reduction of the total alternative space to the relevant *alternative space*. A reduction in the space is defined by the solution domain itself that defines the *constraints* and as such the possible combination of alternatives. The possible alternative space can be further reduced by considering only the combinations of the instantiations that are relevant from the client's perspective and the problem perspective.

Constraints may be defined for the sub-concepts within a concept as well as among higher-level concepts. We describe first the constraints among the sub-concepts within a concept and later among the concepts. *Binary constraints* are the constraints among two concepts. Constraints may be also defined for more than two concepts together. We use the *Object Constraint Language* (OCL) [Warmer & Kleppe 99] that is part of the UML to express the constraints over the various concepts.

Constraint identification is not only useful for reducing the alternative space but it may also help in defining the right architectural decomposition. The existence of many constraints between the architectural components provides a strong coupling and as such it may refer to a wrong decomposition. This may result in a reconsideration of the identified architectural structure of each concept.

Example

From the solution domain we could identify several constraints that restrict the alternative space of the architecture. In the following we will describe examples of the constraints for the sub-concepts of the concepts *Scheduler* and *RecoveryManager* [Weihl 90][Weihl 89][Guerraoui 94] using the Object Constraint Language (OCL). In addition we will provide the constraints among the concepts *Scheduler* and *RecoveryManager*. Figure 4.17 illustrates three constraints for the sub-concepts of *Scheduler*.

1. Conservative Two Phase Locking schedulers need only either a Deadlock Detector or an Infinite Blocking Detector.

```
if  self.SynchronizationScheme.oclIsTypeOf(TwoPhaseLocking)
          and (self.SynchronizationStrategy.oclIsTypeOf(Conservative)
then self.PerformanceFailureDetector.oclIsTypeOf(Deadlock Detector) or
          self.PerformanceFailureDetector. oclIsTypeOf(Infinite Blocking Detector)
endif
```

2. Optimistic and timestamp ordering schedulers need only detectors for either an infinite restart or a cyclic restart.

```
if  self.SynchronizationScheme.oclIsTypeOf(Optimistic)  or
          (self.SynchronizationScheme.oclIsTypeOf(Timestamp Ordering)
then (self.PerformanceDailureDetector.oclIsTypeOf(Infinite Restart Detector) or
          (self.PerformanceFailureDetector.oclIsTypeOf(Cyclic Restart Detector)
endif
```

3. A serial scheduler does not need to detect failures.

```
if  self.SynchronizationScheme.oclIsTypeOf(Serial)
then self.PerformanceFailureDetector.oclIsTypeOf(nil)
endif
```

Figure 4.17 *Constraints for the sub-concepts of Scheduler*

The first constraint defines that for a scheduler with a *two-phase locking* synchronization scheme and a synchronization strategy that is *conservative* either a *deadlock detector* or an *infinite blocking detector* is needed. The reason for this is that the other two performance failures, *infinite restart* and *cyclic restart*, can never occur for this alternative of a scheduler [Cellary et al. 89]. The second constraint indicates that optimistic and timestamp ordering schedulers either need an infinite restart or cyclic restart detector. Finally, the third constraint defines that a serial scheduler does not lead to performance failures because it orders operations of transactions serially and never delays or aborts transactions.

Figure 4.18 illustrates the constraints for *RecoveryManager*. The first constraint defines that a deferred-update recovery technique does not require an undo process in case of restarts. This is because original data objects are not accessed during the execution of transactions and only the copies are affected. The second constraint defines that an update-in-place does not require a redo process. The reason for this is that the original data objects are already accessed during execution of transactions and on commit it is not necessary anymore to install the effects of the transactions.

Figure 4.19 illustrates a constraint among the concept *Scheduler* and *RecoveryManager*. It defines that a serial scheduler will not use a synchronization protocol to provide atomicity, simply because no concurrency is allowed for this scheduler.

```
1.  Deferred-Update does not require undo process
if  self.LogManager.oclIsTypeOf(Deferred-Update)
then self.Restarting.oclIsTypeOf(No-Undo/Redo) or
        self.Restarting.oclIsTypeOf(No-Undo/No-Redo)
endif

2.  Update-In-Place does not require redo process
if  self.LogManager.oclIsTypeOf(Update-In-Place Logging)
then self.Restarting.oclIsTypeOf(No-Redo/Undo) or
        self.Restarting.oclIsTypeOf(No-Redo/No-Undo)
endif
```

Figure 4.18 *Constraints for the sub-concepts of RecoveryManager*

```
1.  A serial scheduler does not synchronize operations for recovery.
if  scheduler.SynchronizationScheme.oclIsTypeOf(Serial)
then recoveryManager.FailureAtomicitySynchronizer.oclIsTypeOf(nil)
endif
```

Figure 4.19 *A constraint between Scheduler and RecoveryManager*

Other constraints are identified for example for the commit and abort protocols of the *TransactionManager*, which must be understood by the different data managers in the system. If the protocols of the *TransactionManager* are changed, then the protocols of the data managers must change accordingly. Due to space limitations we will not further elaborate on the other constraints in this thesis.

4.4.5 Architecture Specification

It consists of the two sub-processes *extracting semantics of the architecture* and *defining dynamic behavior of the architecture.*

Extract Semantics of the Architecture

We consider each concept separately to derive its semantics from the solution domains to provide a more formal, but corresponding, specification. As a format for writing a formal specification we use:

<operation><pre-condition><post-condition>

Hereby, <operation> represents the name of the operation of a concept. The name and the type of each concept variable are described in the part <declarations>. The part <pre-conditions> describe the conditions and assumptions made about the values of the concept variables at the beginning of <operation>. The part <post-conditions> describe what should be true about the values of the variables upon termination of <operation>. Note that this is just one particular way of specifying architectures. For the specification of transaction architectures this type of specification was appropriate, however, other applications may require different specification mechanisms.

Example

Creating and Terminating Transactions

The architecture component *Transaction* represents the application program that is executed as an atomic transaction. We can derive the semantics for this component from the solution domain. For example, the following represents the semantics of *Transaction,* as it has been adapted from [Lynch et al. 94].

```
Transaction::Start
postcondition:
        self.status=”running”

Transaction::Commit
postcondition:
        self.status=”success”

Transaction::Abort
postcondition:
        self.status=”fail”

….
// Additional operations
```

Figure 4.20 *Specification of the interface of Transaction*

A transaction can be started using the operation *Start*, which initializes the transaction parameters and may include application specific operations before the starting the transaction. The variable *status* represents the state of the transaction and can have the values *running*, *success or fail*. The operations *Commit* and *Abort* respectively commit and abort the transaction and may include specific operations after the termination of the transaction. These three operations are generic for most transaction applications. Other operations may be added for specific transaction applications. For example, in a car dealer system, operations such as *Reserve_Car*, *Order_Car* and *Request_CarInfo* would be defined.

Every transaction will be managed by a *TransactionManager* that is basically responsible for the creation, initialization and termination of the transactions. The semantics of *TransactionManager* for managing flat transactions as adapted from [Bernstein et al. 87] is presented in Figure 4.21.

TransactionManager includes the operations *Start*, *RequestCommit*, *Commit* and *Abort*. Further it includes 5 boolean variables *transaction_started*, *transaction_committed*, *transaction_aborted*, *commit_requested* and one variable *transaction* that keeps the *Transaction* object. The operation *Start* initiates the transaction and sets the boolean variable *transaction_started* to true. The initiation of a transaction may include the assignment of, for example, unique transaction id and/or timestamp. A commit of a transaction must always be requested before, which is done by executing the operation *Request_Commit*. This operation is only allowed if the transaction has been started but not completed yet. The operations *Commit* and *Abort* terminate the transaction, set the boolean variables accordingly and initialize the variable *transaction* to nil, so that a new transaction can be started.

```
TransactionManager::Start(T:Transaction)
postcondition:
        self.transaction_started=true;
        self.transaction = T;

TransactionManager::Request_Commit(T:Transaction)
precondition:
        self.transaction_started= true;
        self.transaction_committed = false;
        self.transaction_aborted=false;
postcondition:
         self.transaction_commitrequested= true;

TransactionManager::Commit(T:Transaction)
precondition:
        self.transaction_commitrequested= true;
postcondition:
        self.transaction_committed= true;
        self.transacttion = nil.

TransactionManager::Abort(T:Transaction)
precondition:
        self.transaction_started = true;
        self.transaction_committed = false;
        self.transaction_aborted=false;
postcondition:
        self.transaction_aborted=true;
        self.transaction = nil;
```

Figure 4.21 *Specification of the interface for TransactionManager dealing with flat transactions*

TransactionManager component may also express nested transactions. The specification of additional operations for a transaction manager for nested transactions that we have adapted from the solution domain on nested transaction [Moss 85] is given in Figure 4.22.

The operation *Create_Subtransaction* creates a subtransaction for the corresponding transaction and sets the *parent* of the sub-transaction. A parent transaction is not allowed to complete its own activity until its subtransactions have terminated. This means that the commit and abort operations need to be implemented accordingly. If a subtransaction aborts, the parent can choose different actions, such as ignoring, triggering another transaction or aborting itself. In flat transactions, after the confirmation of a commit from the datamanagers the transaction was able to commit or to abort. In nested transactions, the transaction needs first to report the result of the termination to its parent transaction. In Figure 4.22, the operations *Report_Commit* and *Report_Abort* inform the parent transactions on respectively the commit and the abort of the sub-transaction. Depending on whether open nested transactions or closed nested transactions are implemented the implementation of the commit and abort operations may change accordingly.

TransactionManager::Create_SubTransaction(T:Transaction)
postcondition:
self.subtransactions = self.subtransactions $\cup$\{T\};
T.parent = self.

TransactionManager::getParent()
postcondition:
(**not(**top_transaction **and** self.parent) **or** self

TransactionManager::Report_Commit(T:Transaction)
postcondition:
parent.committed_subtransactions = parent.committed_subtransactions $\cup$\{T\};

TransactionManager::Report_Abort(T:Transaction)
postcondition:
parent.aborted_subtransactions = parent.aborted_subtransactions $\cup$\{T\};

Figure 4.22 *Specification for additional operations of TransactionManager for nested transactions*

Scheduler

The component *Scheduler* deals with the concurrency control of transaction operations in order to keep the data object consistent. Figure 4.23 represents an example of the semantics of the Scheduler that is based on two phase locking. There are five operations, *HandleOperation* for read, *HandleOperation* for write, *CommitRequest*, *Commit* and *Abort*. The operation *HandleOperation* checks whether the operation can be abstracted to a read or write operation. This means that the original operation does not need to be a read or write at all. Subsequently, a check is done on whether the corresponding operation conflicts with previously submitted operations of other transactions. The conflict operation is hereby encapsulated and may depend on different conflict rules for different operations. Before a *Commit* operation can occur first a *CommitRequest* operation must be invoked. A *CommitRequest* operation may also conflict with other operations and therefore this is also explicitly checked. The *Commit* and *Abort* operations result in the release of the locks that have been hold in the sets *readlock_holders* and *writelock_holders* of the corresponding transaction are released.

Scheduler::HandleOperation(T:Transaction, m: Operation)
for *kind*(m) = "read"
precondition:
there do not exist (T’,n) such that (T,m) *conflicts* with (T,m);
postcondition:
self.readlock_holders = self.readlock_holders ∪ (T,m);

Scheduler::HandleOperation(T:Transaction, m: Operation)
for *kind*(m) = "write"
precondition:
there do not exist (T’,n) such that (T,m) *conflicts* with (T,m);
postcondition:
self.writelock_holders = self.writelock_holders ∪ (T,m);

Scheduler::CommitRequest(T:Transaction)
precondition:
there do not exist (T’,n) such that (T,m) *conflicts* with (T,m);
postcondition:
self.commit_requested = self.commit_requested ∪ T;

Scheduler::Commit(T:Transaction)
precondition:
T ∈ self.commit_requested;
postcondition:
self.committed= self.committed ∪{T};
self.readlock_holders = self.readlock_holders - T;
self.writelock_holders = self.writelock_holders - T.

Scheduler::Abort(T:Transaction)
precondition:
postcondition:
self.aborted= self.aborted ∪{T};
self.readlock_holders = self.readlock_holders - T;
self.writelock_holders = self.writelock_holders - T.

Figure 4.23 *Specification of the interface of Scheduler based on Locking*

For the same architectural component *Scheduler* we may derive other semantics from the solution domain on concurrency control. Figure 4.24 defines, for instance, the specification of the operations of a timestamp ordering scheduler [Bernstein et al. 87]. The timestamp ordering scheduler orders conflicting operations according to their timestamps that have been assigned by *TransactionManager*. If two operation p and q are conflicting then the timestamp ordering scheduler processes p before q if *timestamp*(p) < *timestamp*(q).

```
Scheduler::HandleOperation(T:Transaction, m: Operation)
for kind(m) = "read"
precondition:
    there do not exist (T',n) such that if (T',n) conflicts with (T,m)
    and timestamp(T') > timestamp(T)
postcondition:
    self.max_readtimestamp = timestamp(T);

Scheduler::HandleOperation(T:Transaction, m: Operation)
for kind(m) = "write"
precondition:
    there do not exist (T',n) such that (T,m) conflicts with (T,m)
    and timestamp(T') > timestamp(T)
postcondition:
    self.max_writetimestamp = timestamp(T)

Scheduler::CommitRequest(T:Transaction)
precondition:
    there do not exist (T',n) such that (T,m) conflicts with (T,m);
postcondition:
    self.commit_requested = self.commit_requested ∪ T;

Scheduler::Commit(T:Transaction)
precondition:
    T ∈ self.commit_requested;
postcondition:
    self.committed= self.committed ∪{T};

Scheduler::Abort(T:Transaction)
precondition:
postcondition:
    self.aborted= self.aborted ∪{T};
```

Figure 4.24 *Specification of the interface of Scheduler based on timestamp ordering*

RecoveryManager

Figure 4.25 represents a specification of the interface of RecoveryManager that has been adapted from the solution domain on recovery [Bhargava 87]. In this example, the RecoveryManager has 5 operations. The operation *HandleOperation* will either log the operation or the state of the data object that is being accessed. The operation *Commit* makes the effect of the transaction persistent by storing this in stable storage. The operation *Abort* rollbacks the effects of the transaction by using the logged information. The operation *Restart* will be invoked in case of system failures. This operation uses the logged information to undo the effects of the aborted or active transactions and redo the effects of the committed transactions that have not been made persistent yet. Finally, the operation *Checkpoint* is regularly invoked to make a snapshot of the system so that the *Restart* operation is optimized. In Figure

4.25 only a generic interface for recovery is presented. However, the semantics for each of the different variations on these recovery protocols can be easily derived from the solution domains.

RecoveryManager::HandleOperation(T:Transaction, m: Operation)
postcondition:
operation (T,m) or the accessed object state is logged;

RecoveryManager::Commit(T:Transaction)
postcondition:
make effects of transaction T persistent;

RecoveryManager::Abort(T:Transaction)
postcondition:
undo effects of transaction T;

RecoveryManager::Restart()
postcondition:
undo effects of active aborted transactions
redo effects of committed transactions;

RecoveryManager::Checkpoint()
postcondition:
store the state of the system in stable storage.

Figure 4.25 *Specification of the operations of RecoveryManager*

PolicyManager

An example specification for (a part of) the interface of the PolicyManager component is given in Figure 4.26. *PolicyManager* is responsible for dynamic adaptation of the transaction protocols based on selected performance parameters such as transaction throughput, transaction response time, transaction blocking ratio and transaction restart ratio [Agrawal 87]. The operations *AddParameter* and *RemoveParameter* respectively add and remove a performance parameter from the set *performanceParameters*. The operation *ReadSystemParameters* senses the system and derives the values for the parameters in the set *performanceParameters*. These values are used by the operation *ChooseTransactionProtocols* to determine appropriate transaction protocols such as scheduling and recovery algorithms. Finally, the operation *DeterminePolicy* defines the policy for the dynamic adaptation mechanism. The choice of transaction protocols may not always be defined by the system characteristics but the transaction or data characteristics may also impose some constraints on the selection of the transaction protocols. For example, for long transaction a locking scheduler may be preferred over an optimistic scheduler [Agrawal 87]. Large binary data objects may prefer to adopt operation logging techniques instead of image logging to optimize the memory space [Elmagarmid 92]. *PolicyManager* must therefore balance between these different wishes of the transaction programmer, the system performance parameters and the data object characteristics.

PolicyManager::AddParameter(P: PerformanceParameter)
postcondition:
    self.performanceParameters = self.performanceParameters $\cup$ P;

PolicyManager::RemoveParameter(P: PerformanceParameter)
postcondition:
    self.performanceParameters = self.performanceParameters - P;

PolicyManager::ReadParameterValues()
postcondition:
    self.performanceParameters determined;

PolicyManager::ChooseTransactionProtocols(T:Transaction)
postcondition:
    T.transactionProtocols determined;

PolicyManager::DeterminePolicy()
postcondition:
    self.policy = *priority*(user, system, data).

Figure 4.26 *Specification of the interface of PolicyManager*

Correctness of Transaction Semantics

We have shown that the semantics for the components of the atomic software architecture can be derived from the solution domains and gave some examples of the semantics of the architectural components of the atomic transaction architecture. Besides of the rich semantics that we could derive from the solution domain an important issue is whether the provided semantics is correct. In this thesis we will not provide the correctness proofs but refer to the related literature on atomic transactions. For example, in [Lynch et al. 94] the I/O automaton model is described, which is a formal model for modeling concurrent, and distributed systems. Hereby, each system component, concept or technique is analyzed and expressed as an automaton, a mathematical object with states and named transitions between them[35]. The actions of the automaton can be classified as *input*, *output* or

[35] This is almost similar to a non-deterministic finite-state automaton. One difference is that in the I/O automaton model an automaton need not be finite-state, but can have an infinite state set.

internal. The input actions represent events from the environment, the output actions represent events that components performs itself and finally the internal actions represent the events internal in a component that are not externally observable (such as changing a local variable). For each automaton the actions are described with an *action signature*. An automaton can put restrictions on when it will perform an output or internal action, but is unable to restrict input actions.

The correctness criteria for a concurrent system that is modeled as automata are expressed as restrictions on the sequences of actions that are part of the interface of the data items and its users. The basic assumption is that a sequence of actions is correct if it can be generated by a serial system.

An automaton is defined as a tuple consisting of four components [Lynch et al. 94]:

- an action signature *sig(A)*
- a set *states(A)* of **states**
- a nonempty set *start(A)* $\subseteq$ *states(A)* of **start states**, and
- a **transition relation** *steps(A)* $\subseteq$ *states(A)* x *acts(sig(A))* x *states(A)*, with the property that for every state s' and input action π there is a transition (s', π, s) in *steps(A)*.

In [Lynch et al. 94] states are generally determined by giving values to a collection of variables. Further, the transition relations of an automaton are not described by listing all its elements as triples but rather a simple specification language is used where an *effect* is described for each action and a *precondition* for each local action.

Using this model the authors formalize and analyze transaction processing theories, serializability, logging, locking, nesting, timestamping etc. For a more detailed description of this automaton model we refer to [Lynch et al. 94]. We use this model to proof the semantics of the architectural concepts that we derived. It follows that since we derive the abstract semantics from the solution domain, the link to the formal models is easily identified and we can utilize these to validate the software architecture. For this, we map each architectural concept to an automaton and define the operations in the specification of the concept as *internal* and *input* actions of the newly identified automaton. To define the *output* actions we look for the architectural concepts that the corresponding concept communicates with and define the operations of other concepts that are invoked as the output operations of the automaton. These operations together will provide the complete action signature of the automaton. Consider for example the specification of the concept *TransactionManager* as presented in Figure 4.21. We could define an automaton called *TransactionManager* that includes the operations as defined in Figure 4.21. Since all of these operations are invoked by other components we map these to *input* operations of the automaton. There are no internal operations. The output operations can be identified in the specification of the concept *Transaction* in Figure 4.20 and this completes the action signature of the automaton *TransactionManager*. Subsequently the start states of the automaton and the transition relations can be relatively easily defined. Once the automaton is described we use it to continue with the correctness proofs of the adopted transaction semantics. Since many publications exist on correctness proofs of the semantics of transaction protocols [Papadimitriou 86][Cellary et al. 89][Bernstein et al. 87] and it is not our goal to extend the transaction theory we will not elaborate on this issue in this thesis.

Define Dynamic Behavior of the Architecture

The specifications of the architectural components are used to model the dynamic behavior of the architecture. For this purpose we use the so-called collaboration diagrams which are interaction diagrams to illustrate the dynamic view of a system [Booch et al. 99]. Collaboration diagrams show the structural organization of the components and the interaction among these components. We derive the collaboration diagrams from the pre-defined specifications of the architectural concepts.

Example

Figure 4.27 represents an example of a collaboration diagram for the atomic transaction architecture. The components in the collaboration diagram represent instances of the architectural components, which is represented by a double colon preceding the name of the architectural component. The flow of control is represented by means of directed arrows that are labeled with messages. To indicate the temporal sequencing the messages are numbered. The collaboration diagram shows the interactions for starting, handling operations, committing and aborting transactions.

The messages with the sequence number 1 are part of the scenario for starting a transaction. A transaction is started by the object t:TransactionApplication that sends a start operation to the object tm:TransactionManager. The tm object informs the starting of the new transaction to the object pm:PolicyManager that reads the values of the performance parameters and chooses the appropriate transaction protocols for the transaction.

The messages with sequence numbers 2 define a scenario for handling transaction operations. After the transaction has started, the operations that are send by the object t:aTransactionApplication will be captured and handled by the object tm:TransactionManager. The operation will be forwarded to the policy manager and the data manager object. The object dm:DataManager will request the scheduler and the recovery manager object to provide a decision on the acceptance or reject of the operation. If the operation is allowed to execute then it will be dispatched to the atomic object.

Finally, the messages with the sequence numbers 3 and 4 define respectively the scenarios for committing and aborting transactions. The control flow for these scenarios can be easily derived from Figure 4.27.

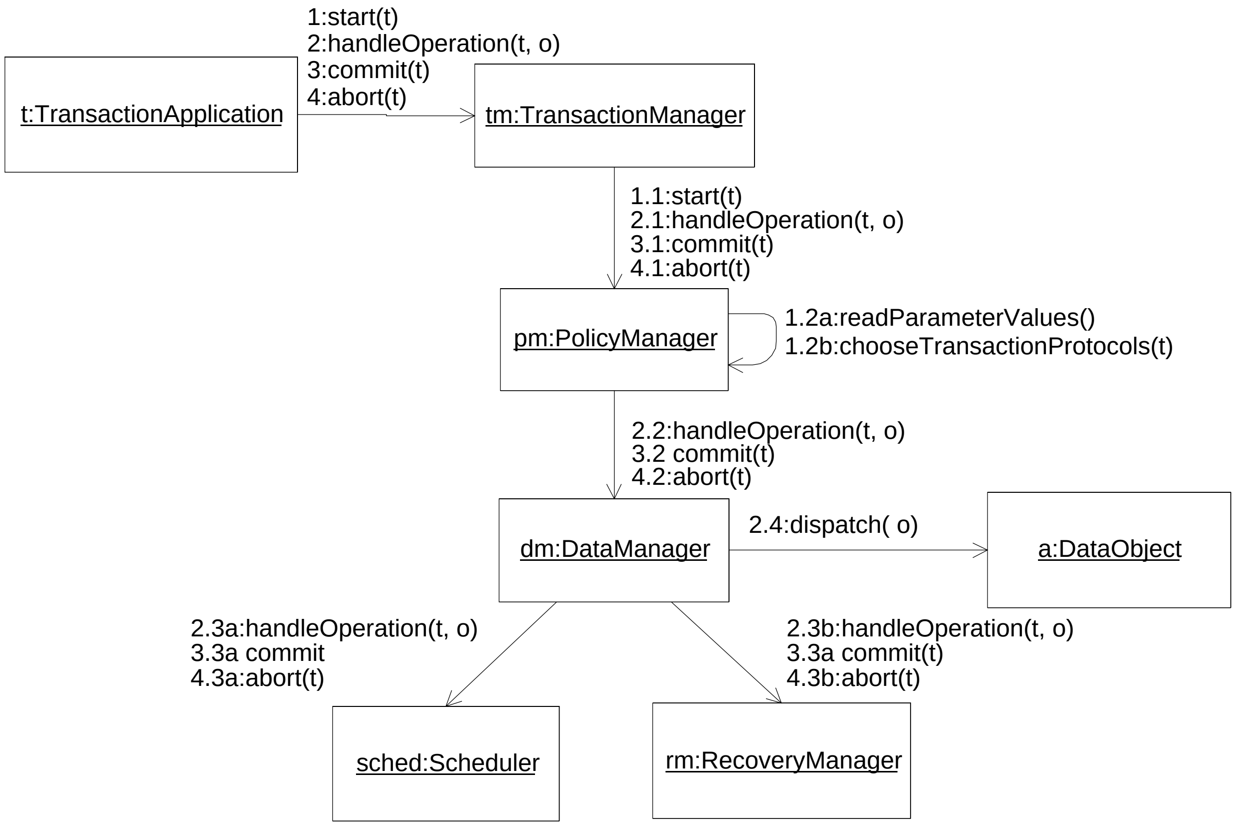


Figure 4.27 *Collaboration diagram for the atomic transaction software architecture*

4.5 Discussion and Conclusions

In this chapter we presented the *synthesis-based software architecture design approach*. This approach is derived from the concept *synthesis* of mature engineering disciplines whereby the initial problem is decomposed into sub-problems that are solved separately and later integrated in the overall solution. During the synthesis process design alternatives are searched and selected based on the existing solution domain knowledge.

An important issue in software architecture design is to find the right abstractions and the adequate leveraging of the architecture. The novelty of the *synthesis-based software architecture design approach* with respect to the existing architecture design approaches is that it makes the processes of *problem analysis*, *solution domain analysis* and *alternative space analysis* explicit. During the *problem analysis*, the client requirements are mapped onto the technical problems providing a more objective and reliable description of the problem. During the *solution domain analysis*, stable architectural components with rich semantics are derived from the solution domain concepts that are well-defined and stable themselves. The solution domain analysis itself is leveraged by the pre-identified technical problems so that the right detail of the solution domain model is ensured. The *alternative space analysis* explicitly depicts the possible set of design alternatives that can be derived from the architectural components.

We have illustrated the approach by applying it to the design of an atomic transaction architecture for a distributed car dealer system in a project of Siemens-Nixdorf. Apart from this, experimental studies have been carried out with earlier versions of this approach in pilot studies that were carried out by MSc students. For example, in [Vuijst 94], a software architecture for image algebra was derived for the laboratory for clinical and experimental image processing. The basic solution domain for this architecture was image algebra and several related publications could be identified from which sufficient stable abstractions were derived for the design of the software architecture. The atomic transaction and the image algebra domain appeared to be examples of well-defined and sufficiently formalized domains. The experimental studies have been, however, also applied on domains that are less formalized. In [Arend 99], for example, a software architecture has been derived for a Quality Management Systems for efficient information retrieval and in [Willems 98], a software architecture has been derived for insurance systems. In both cases, several publications could be identified on the corresponding domains, but in addition it was also necessary to refer to the factual knowledge and experiences for the design of the software architecture. The solution domain may thus consist of a combination of various forms of solution techniques such as theories, solution domain experts, and experiences in the corresponding domain.

In the following we will list the conclusions that we could obtain from our experience in applying the synthesis approach to the project on atomic transactions.

1. *Explicit mapping of requirements to technical problems facilitates the identification and leveraging of the necessary solution domains.*

After our requirements analysis and technical problem analysis processes as defined in sections 4.4.1 and 4.4.2 respectively, it appeared that the given client requirements did not fully describe the right detail of the desired problem. The basic requirement was to provide adaptable transactions protocols that were derived from the various expected needs of different dealers in different countries. From the initial requirement specification, however, it followed that with adaptability of transaction protocols it was only referred to a restricted number of concurrency control protocols. During the problem analysis phase we generalized this requirement to the adaptation of various transaction protocols including transaction management, concurrency control, recovery and data management techniques. After interactions with the client and a study of the car dealer distribution system it appeared that many transaction protocols were relevant although they had not been explicitly mentioned in the requirement specification. We observed that the technical problem identification is an iterative process between the technical problem analysis and solution domain analysis processes.

On the one hand, we directed and leveraged our solution domain analysis using the identified technical problems. Since every (sub-)problem corresponds only to a restricted set of solution domain we did not need to consider the whole solution domain space at once. For example, for the concept *DataManager* we did not need to consider version management and replication management because this was deliberately put out of the scope of the project. For the concept *Scheduler* we ruled out the solution domain that dealt with semantic concurrency control techniques. The identified technical problems provided us the means where to search or not to search for the solution domain.

On the other hand, the technical problems could be better defined after the solution domains were better understood. For example, only after a solution domain analysis on concurrency control, as described in section 4.4.3, we were better able to accurately define the sub-problems related with the concept *Scheduler*. This observation may imply that for the problem analysis phase one may require a domain engineer who is an expert on the corresponding domain and knows the different technical problems that are related to the domain. In our example project typically a transaction domain expert at the early phase of problem analysis would be of much help.

2. *Solution domain provides stable architectural abstractions*

The synthesis-based approach provides an explicit solution domain analysis process for identifying the right abstractions. After analyzing and comparing the solution domain on transaction theory it appears that it is rather stable and does not change abruptly but only shows a gradual specialization of the transaction concepts. Because the solution domain is stable it provides a reliable source for providing stable architectural abstractions. In the solution domain analysis process as described in section 4.4.3 we illustrated how we could derive stable concepts for the design of the atomic transaction architecture. We were able to derive both the overall architecture and refine the architectural concepts to the required detail level.

The requirement of stable solution domains in the synthesis-based approach implies that a given problem can only be solved to the extent that it has been explored in the solution domain. If it appears that the solution domain is not well-established the software engineer may decide to terminate the synthesis process, reformulate the technical problem or initiate a research on the solution domain. The latter decision shows that the synthesis process may provide important input for the scientific research because it may indicate the issues that need to be resolved in the corresponding solution domains[36].

3. *Solution domains provide rich semantics for realization and verification of the architecture.*

Solution domains not only provide stable abstractions but in addition these abstractions have rich semantics which is important for the realization and verification of the software architecture. As described in section 4.4.5 on architecture specification, we could derive rich semantics for the architectural abstractions directly from the solution domain knowledge of atomic transactions. We have illustrated this process for various components in the atomic transaction architecture.

The solution domain is not only useful for deriving architectural abstractions, but in addition it is also a reliable source for validating the correctness of the developed architecture. We were able to identify many publications that explicitly deal with correctness proofs of various transaction protocols. We validated the architectural components and their semantics by utilizing these knowledge sources.

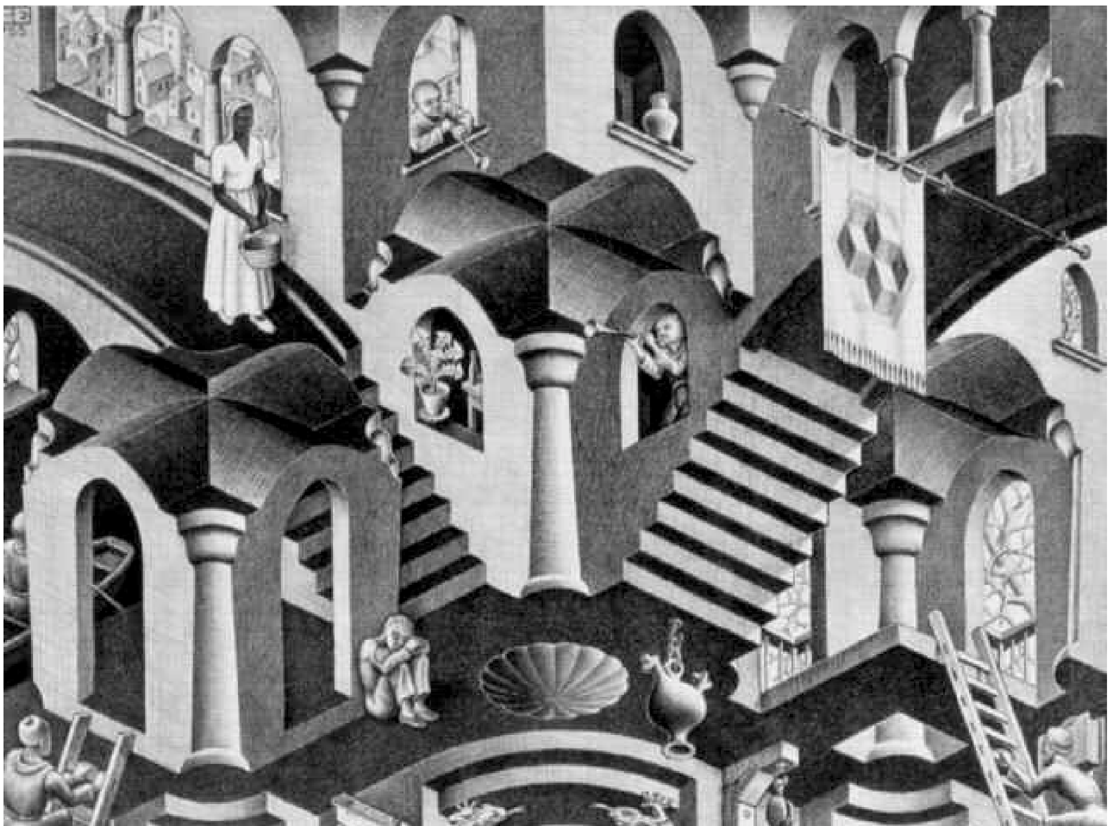
[36] Note that this represents an example of the interaction between engineering and scientific research

4. *Adaptability of an architecture can be determined by an explicit alternative space analysis of the solution domain.*

In the synthesis-based software architecture design approach, alternative space analysis is an explicit process. Thereby, for each concept the set of alternatives are described and constraints are defined among these alternatives. This together results in a depiction of the set of possible alternative designs or alternative space, that may be derived from the given software architecture. As described in section 4.4.4 we have, for instance, defined the alternatives for the concepts *Scheduler* and *RecoveryManager*. From the solution domain analysis we extracted the constraints within each of these concepts and constraints that apply among alternatives of these concepts. We had two problems in the alternative space analysis process for the example project. First, although we have derived the conceptual architectures from the solution domain itself, during the alternative definition process it followed that not all the alternatives were explicitly described in the literature. For example, for the concept *Scheduler* we could identify only around 10-15 scheduler types that were described in the literature. The other alternatives are primarily seen as variations of these basic scheduler types. In our approach we could depict every single alternative explicitly. The second problem that we encountered was that the constraints within and among the alternatives of the concepts are generally not explicitly stated in the literature and finding these is very time-consuming. Defining constraints of solution domain concepts requires the full understanding of these concepts. The existence of an explicit description of these constraints may indicate the maturity level of the corresponding solution domain. It appears that the transaction literature has many well-established concepts and we could also identify some publications that explicitly dealt with the constraints among the concepts, however, this is not the case for all the concepts.

Chapter 5

Balancing Architecture Implementation Alternatives


M.C.Escher's Concave and Convex © 2000 Cordon Art-Baarn-Holland. All rights reserved.

M.C. Escher - Concave and Convex

What we see and what we understand depends on our perspective. In "Concave and Convex," Escher has created a paradoxical world where concave and convex are constantly shifting, throwing the mind into complete ambiguity and confusion. Three little houses stand near one another, each under a crossvaulted roof. We have an exterior view of the left-hand house, an interior view of the right-hand one and an either exterior or interior view of the one in the middle, according to choice. The cluster of cubes on the flag announces the basic visual motif of the composition. Escher plays with the ambiguity of volumes on the flat picture plane; they switch from solid to hollow, from inward to outward, from roof to ceiling, like the cubes in the flag.

Software Architecture Design Analogy

What we see and what we understand from a given software architecture depends on our perspective. If the software architecture is not well-defined it may be ambiguous and even self-contradictory. This will confuse the stakeholders of software architecture and its realization will be a complicated task.

5.1 Introduction

"Indeed, this subtle and complex freedom from inner contradictions is just the very quality which makes things live".
- Christopher Alexander, The Timeless Way of Building

The architecture design phase may be followed by an object-oriented analysis and design phase in which a set of heuristic rules are provided to guide software engineers to analyze, design and implement object-oriented software systems. Since architecture specifications are generally abstract, one may derive various different implementation alternatives for the same architecture specification. Each alternative may have different adaptability, performance and reusability characteristics. Object-oriented analysis and design methods help software engineers to express their solutions in terms of object-oriented abstractions. However, a number of problems may be encountered in deriving the alternatives from the architecture specification:

Firstly, it is generally difficult for software engineers to explicitly identify the different alternatives of a design. Secondly, in current methods the relation between object-oriented designs and quality factors seems to be more implicit than explicit. This is because object-oriented methods rely on the intrinsic quality factors of the object-oriented abstractions rather than considering quality factors as explicit design concerns. Finally, current methods do not provide adequate techniques to balance various quality factors, such as adaptability and performance. Alternative designs are rarely selected for ultimate adaptability or performance but rather it is a compromise of multiple considerations. At almost every stage of the software development lifecycle, software engineers have to cope with various design alternatives. Of course while defining object models, software engineers apply their knowledge and experience. They generally compare the alternatives based on their intuition. This is, however, an implicit process.

This chapter introduces a new formalism called *Design Algebra*, which is used to depict the space of design alternatives, and define design rules for comparing, evaluating and composing them. The techniques provided by design algebra can be integrated within the object-oriented methods. The applicability of Design Algebra is illustrated using the transaction system example.

This chapter is organized as follows. The next section introduces an example and explains the problems addressed in this paper. Section 3 presents the principles of design algebra, and illustrates its applicability first formally and then intuitively using the example problem. Section 4 shows how various quality factors such as adaptability and time performance can be balanced. Evaluation of the approach is given in Section 5. Section 6 refers to the related work. Finally, section 7 gives conclusions.

5.2 The Problem Statement

In this section we will describe the problem statement using the atomic transaction architecture that we developed in chapter 4. Section 5.2.1 presents an example problem for deriving design alternatives from the architecture concepts *DataManager* and *Scheduler*. Section 5.2.2 explains the problems addressed in this paper.

5.2.1 Example: Designing Alternative Schedulers

Figure 5.1 shows a part of the atomic transaction architecture.

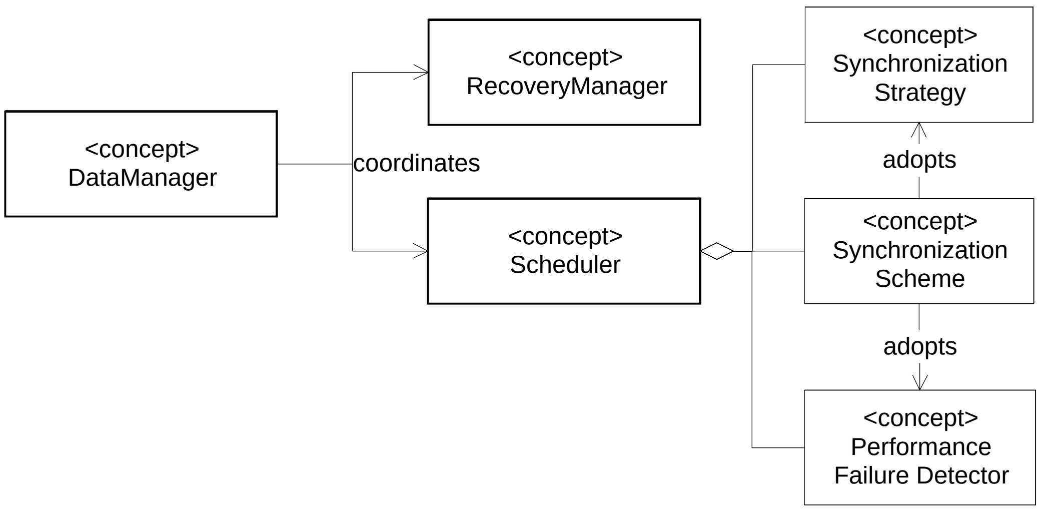

<concept> DataManager
coordinates
<concept> RecoveryManager
<concept> Scheduler
<concept> Synchronization Strategy
adopts
<concept> Synchronization Scheme
adopts
<concept> Performance Failure Detector

Figure 5.1 *Conceptual architecture for part of the Atomic Transaction Architecture*

The *DataManager* uses the functionality of the *Scheduler* and the *RecoveryManager* to preserve the consistency of the data objects that it manages. During the alternative space analysis of the synthesis-based software architecture design process, the set of alternatives in the solution domain of the *Scheduler* and *RecoveryManager* has been explored and described.

After the architecture definition we may focus on deriving analysis and design models from this architecture. For this purpose, we may utilize object-oriented modeling that provides abstractions such as inheritance or aggregations, classes or operations, etc. Realizing the architecture means that the solution domain concepts need to be represented using these object-oriented abstractions.

5.2.2 Problem Description

The problems that we address in this chapter can be grouped under three related categories: utilization of design space, designing alternatives based on quality factors and balancing the quality factors. These are explained in the following three subsections.

Utilization of the Design Space

Considering the examples in the previous section, we observe that realizing an architecture in a design involves the consideration of many design alternatives. A *design space* is informally defined as a set of all the possible design alternatives for a given design problem. Object-oriented analysis and design methods provide several abstractions to define various kinds of alternative models suitable for different stages of software development. The various alternative abstractions enable the software engineer to derive various alternative designs from the same architecture.

Let us explain this in more detail by considering some design alternatives that can be derived from the architecture of Figure 5.1. We consider the top-level concepts *DataManager* and *Scheduler*. In the design alternative of Figure 5.2, the concept *DataManager* has been mapped to a class *DataManager* and the concept *Scheduler* has been mapped to an operation *schedule(Message)* of the class *DataManager*. The sub-concepts *Synchronization Scheme*, *Synchronization Strategy* and *Performance Failure Detector* are all hidden in the implementation of the operation *schedule*.

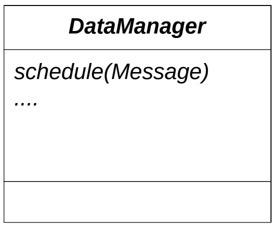


Figure 5.2 *Mapping the concept* Scheduler *to a single operation.*

In Figure 5.3, the concept *DataManager* and the concept *Scheduler* are both mapped to the object-oriented class concept and the *Scheduler* is defined as a part of the *DataManager*. For the Scheduler an inheritance-based design is chosen where an abstract class *UniversalScheduler*[37] has been introduced, which declares the necessary interface for its subclasses *TwoPhaseLocking*, *OptimisticScheduler*, and *TimestampOrderingScheduler*. Alternatives of *Scheduler's* sub-concept *Synchronization Scheme* have been mapped to the various sub-classes of *UniversalScheduler*. The sub-

[37] The Unified Scheduler abstraction was inspired from [Campbell et al. 93]

concepts *Synchronization Strategy* and *Performance Failure Detector* are hidden in the implementation of the operations of the Scheduler classes.

These two design models are not the only alternatives and actually a considerable number of design alternatives may be derived from the same architecture. Depending on our design choices and our consideration for, e.g. the granularity of the required changes, different designs of the architecture may be derived. We may use a separate class for each sub-concept, define these as abstract methods, map these to single methods etc. Current object-oriented analysis and design methods, however, do not provide adequate means to identify and describe the possible design alternatives, i.e. the design space. As a matter of fact, while designing object models, software engineers apply their knowledge, experience and intuition to compare the design alternatives. This process, however, is rather implicit and lacks explicit support. Without knowledge of the design space it is difficult to specify, compare and prioritize the design alternatives. We maintain that object-oriented methods should provide explicit means to determine and reason about the design space and the individual alternatives.

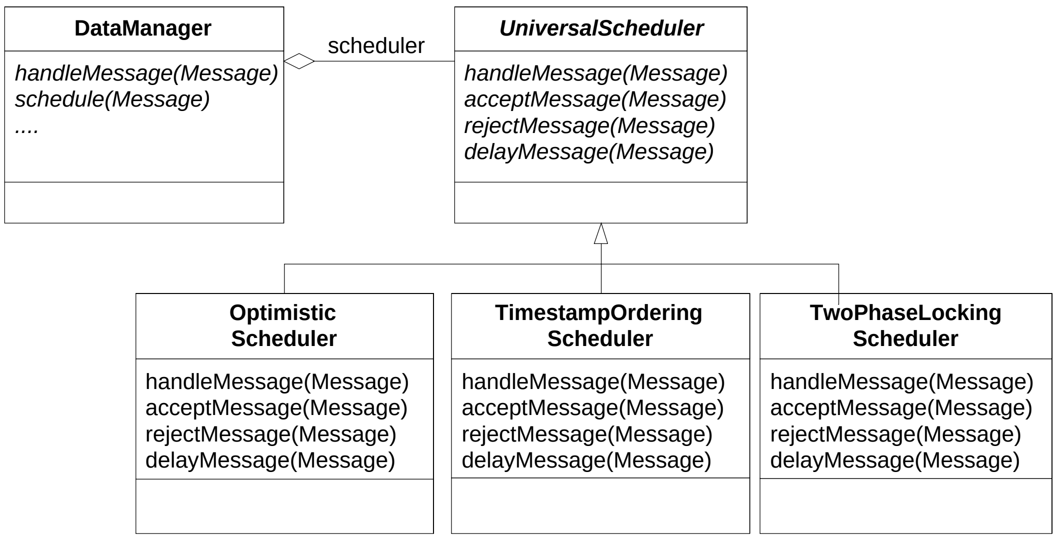


Figure 5.3 *An alternative object-oriented design of the part of the architecture using inheritance.*

Designing alternatives based on quality factors

Design alternatives can be compared with each other based on certain quality factors such as adaptability, performance and reusability. For example, the alternative in Figure 5.2 provides a straightforward implementation of an operation *schedule* in the class *DataManager* and results in a higher time performance compared to the alternative designs of Figure 5.3, that map the Scheduler to classes using inheritance.

On the contrary, the inheritance-based alternative in Figure 5.3 provides a higher reusability than the alternative of Figure 5.2 because every concept is separately represented in each sub-class. If it is required to increase the dynamic adaptability then an alternative model as presented in Figure 5.4 may be designed. Here, every sub-concept is an aggregation and separately represented as a class. Obviously, we can extend this reasoning and derive different alternatives that will perform differently for the various quality factors.

In the analysis and design processes it is important to consider the quality factors such as adaptability, performance and reusability explicitly rather than evaluating these factors after the delivery of programs. In current object-oriented analysis and design methods, however, the relation between object-oriented designs and quality factors seems to be more implicit than explicit. This is because object-oriented methods rely on the intrinsic quality factors of the object-oriented abstractions rather than considering quality factors as explicit design concerns.

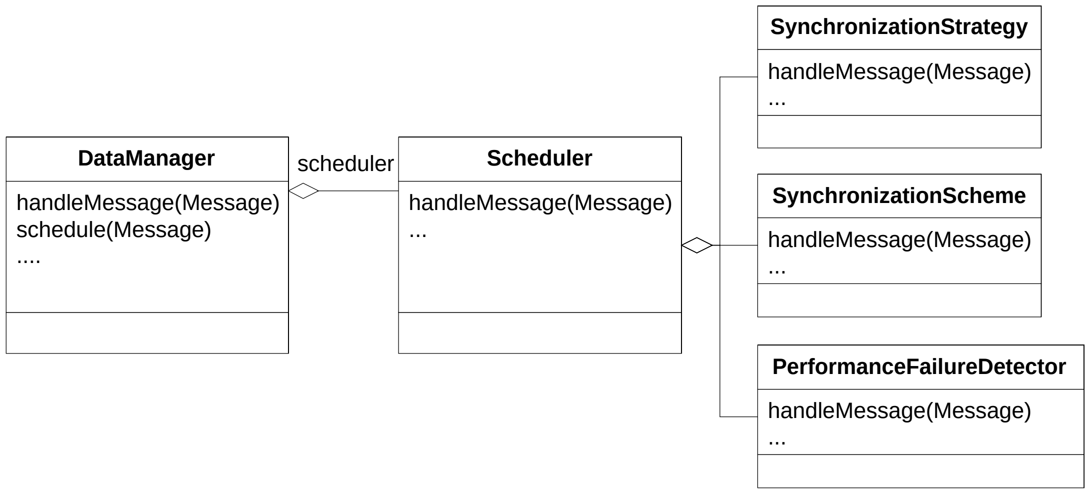


Figure 5.4 *An alternative object-oriented design of the part of the architecture using aggregation*

Balancing quality factors

While designing software systems, software engineers have to carefully balance various quality factors. Software is rarely designed for ultimate adaptability, performance or reusability but rather it is a compromise of multiple considerations. In general there are many correct solutions for the same problem. In the example problem, one may identify many alternative designs, which will differ with respect to adaptability, performance and reusability factors. Providing ultimate adaptability may create too much overhead. Aiming at fastest implementation may result in unnecessarily rigid software. Aiming at most reusable software may introduce redundant abstractions for a given problem. Software engineers, therefore, must be able to compare, evaluate and decide between various alternatives based on the relative importance of the quality factors.

5.3 Design Algebra

In this section we define a formalization technique called *design algebra*. An *algebra* is a formal structure consisting of *sets* and *operations* on those sets. *Relational algebra* is an algebra for manipulating relations. Design Algebra is an application of relational algebra for modeling design spaces and balancing design alternatives.

5.3.1 Notion of Design Space

In the previous section we have provided an intuitive definition of design space[38] as a set of all the possible alternatives for a given design problem. In this section we will elaborate on the notion of design space and describe it in more detail.

We define a ***design space*** as a multi-dimensional space from which the set of alternatives for a given design problem can be derived. We may define a design space for every concept in a solution domain. The design space is spanned by an independent set of ***dimensions***[39]. The dimensions are represented by the sub-concepts of the concept in the solution domain. Consider, for example, the concept *Scheduler* that can formally be described as follows:

$$M_{Scheduler} = (Sch, Str, PFD)$$

Here, *Sch*, *Str* and *PFD* represent the sub-concepts *Synchronization Scheme, Synchronization Strategy* and the *Performance Failure Detector*, respectively. A design space for *Scheduler* consists therefore of three dimensions, which are represented by these sub-concepts.

As described in the previous chapter the concepts are defined by a solution domain analysis process, which involves collecting the related information from various sources, and detecting the commonalties among them through comparison. These

[38] The concept *space* is a well-known term in mathematics and is often used to define a *metric space, topological space* or *vector space.*

[39] Compare this to the notion of canonical basis of a vector space in linear algebra. Any m-dimensional vector in vector space V can be constructed as a linear combination of the vectors $x_1, x_2, \ldots, x_m$ if those vectors are independent. A set of m vectors which span a space V of dimension m are said to form a *basis* for that space.

common abstractions generally correspond to the fundamental concepts in that domain. Concepts form well-defined and stable abstractions and as such are very suitable to represent the independent dimensions of a design space.

Every dimension has a set of ***coordinates*** that are elements of a coordinate set. In design algebra, the coordinate set represents a ***property set*** that represent various ***properties*** that may be assigned to the sub-concepts. An example of a property set may be the concepts of the object-oriented model. In the object-oriented model, a concept can be represented either as a class, an operation or an attribute. Therefore we may define the property set *Object* as follows:

$$P_{Object} = (CL, OP, AT)$$

The ***degree of a dimension*** represents the total numbers of the properties. A ***design alternative*** represents a point in the design space.

Figure 5.5 shows the graphical representation of an example of a design space that utilizes the sub-concepts of *Scheduler* as dimensions and the coordinates of these dimensions are the properties of the set P_{Object}. The design space is named $S_{ObjectScheduler}$.

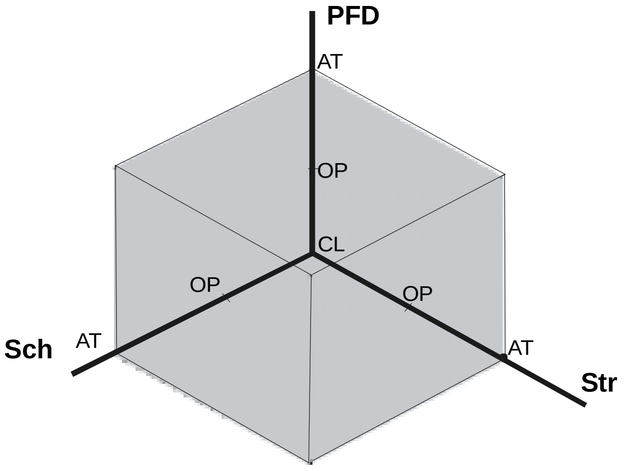


Figure 5.5 *Design Space of Scheduler with Object as property set*

Since this design space has 27 points this implies that there are in total 27 theoretically possible design alternatives within this space. An alternative in this design space is for example *((Sch,CL) (Str,CL) (PFD,CL))* which represents the selection of the sub-concepts of *Scheduler* as classes[40].

[40] Note that not all the 27 alternatives in this design space may be possible or desired. In later sections we will describe the techniques for reducing the design space according to these constraints.

If adaptable models are required then *Adaptability* needs to be considered as a property set. Adaptability can be defined as the ease of changing an existing model to new requirements. To this aim, we have to deal with two contradictory goals: On one hand we have to fix the concepts for robustness and time performance. On the other hand, we need to make concepts adaptable for flexibility [Tekinerdogan 97]. Adaptability can be defined either at compile-time or at run-time. The property dimension *Adaptability* may then be defined as follows:

$$P_{Adapt} = (FX, ADc, ADr)$$

Hereby, the three dimension values *FX*, *ADc* and *ADr* qualify concepts as fixed, compile-time adaptability and run-time adaptability respectively.

Figure 5.6 represents the graphical representation of the design space $S_{AdaptScheduler}$ that is also a three-dimensional space consisting of the dimensions Sch, Str and PFD. The coordinate set is now defined by the properties of P_{Adapt}.

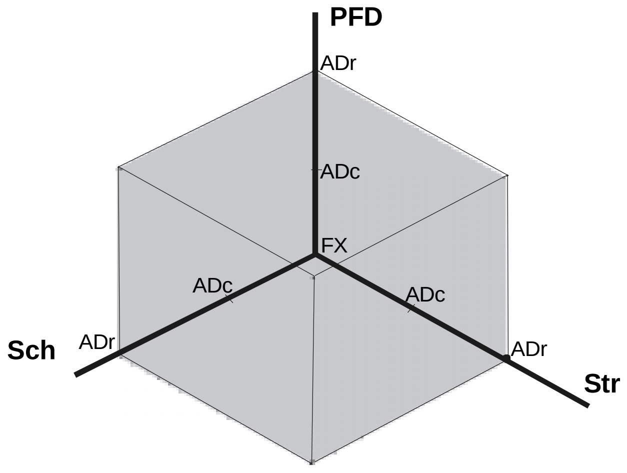


Figure 5.6 *Design Space of Scheduler with* Adaptability *as property set*

5.3.2 Formalizing Design Space Composition

In design algebra, design spaces are defined as function spaces that map concepts to properties. The design space $S_{ObjectScheduler}$, may be defined in the following formula:

$$S_{ObjectScheduler} :: M_{Scheduler} \rightarrow P_{Object}$$

This function defines the alternative design space that includes the following 27 theoretically possible alternatives:

$S_{ObjectScheduler}$ =

{ ((Sch,CL) (Str,CL) (PFD,CL)), ((Sch,CL) (Str,CL) (PFD,OP)), ((Sch,CL) (Str,CL) (PFD,AT)),
((Sch,CL) (Str,OP) (PFD,CL)), ((Sch,CL) (Str,OP) (PFD,OP)), ((Sch,CL) (Str,OP) (PFD,AT)),
((Sch,CL) (Str,AT) (PFD,CL)), ((Sch,CL) (Str,AT) (PFD,OP)), ((Sch,CL) (Str,AT) (PFD,AT)),
((Sch,OP) (Str,CL) (PFD,CL)), ((Sch,OP) (Str,CL) (PFD,OP)), ((Sch,OP) (Str,CL) (PFD,AT)),
((Sch,OP) (Str,OP) (PFD,CL)), ((Sch,OP) (Str,OP) (PFD,OP)), ((Sch,OP) (Str,OP) (PFD,AT)),
((Sch,OP) (Str,AT) (PFD,CL)), ((Sch,OP) (Str,AT) (PFD,OP)), ((Sch,OP) (Str,AT) (PFD,AT)),
((Sch,AT) (Str,AT) (PFD,CL)), ((Sch,AT) (Str,AT) (PFD,OP)), ((Sch,AT) (Str,AT) (PFD,AT))
((Sch,AT) (Str,OP) (PFD,CL)), ((Sch,AT) (Str,OP) (PFD,OP)), ((Sch,AT) (Str,AT) (PFD,AT))
((Sch,AT) (Str,AT) (PFD,CL)), ((Sch,AT) (Str,AT) (PFD,OP)), ((Sch,AT) (Str,AT) (PFD,AT)) }

In the graphical representation of Figure 5.5 each of these alternatives appeared as a point in the space.

The total number of alternatives that can be extracted from $S_{ObjectScheduler}$ can be computed as follows:

$$numberOfAlternatives(S_{ObjectScheduler}) = size(P_{Object})^{size(M_{Scheduler})} = 3^3 = 27$$

The function *size* returns the size of the tuples of the dimension. The function *numberOfAlternatives* computes the number of alternatives of the design space.

The above formulas can be easily generalized. Assume that we have a solution domain concept M_{Domain} consisting of the sub-concepts $c_1, c_2, \dots, c_n$ and a property set $P_{Property}$ consisting of the properties $p_1, p_2, \dots, p_m$. The design space $S_{PropertyDomain}$ can be defined as follows:

$$S_{PropertyDomain} :: M_{Domain} \rightarrow P_{Property}$$

Alternatives of this space consist of a set of n tuples with 2 elements, one from M_{Domain} and one from $P_{Property}$. The size of the design space $S_{ObjectDomain}$ and the number of alternatives that can be derived from this design space can be computed using the following formulas:

$$numberOfAlternatives(S_{propertyDomain}) = size(P_{property})^{size(MDomain)} = \mathrm{m}^{\mathrm{n}}$$

In the same manner we can define the design space $S_{AdaptScheduler}$ that is defined as a function space mapping the set M_{Domain} to the set M_{Adapt} and includes the set of domain alternatives with the adaptability properties:

$$S_{AdaptScheduler} :: M_{Scheduler} \rightarrow P_{Adapt}$$

This design space determines the possible adaptability properties for every sub-concept of *Scheduler*. An alternative that can be selected from this space is, for instance, *((Sch, ADc), (Str,ADc), (PFD,ADr))*. Within this alternative the synchronization scheme (*Sch*) and the synchronization strategy (*Str*) has been selected as compile-time adaptable and the performance failure detector *(PFD)* as run-time adaptable. The total number of alternatives that can be selected from this space equals $3^3 = 27$ alternatives.

We may also define more complex design spaces by defining function spaces that map a concept to more than one property set. For example the following represents a design space for adaptable object models for schedulers:

$$S_{AdaptObjectScheduler} :: M_{Scheduler} \rightarrow (P_{Adapt} \times P_{Object})$$

Note that this function is equivalent to the pair of functions ($M_{Scheduler} \rightarrow P_{Adapt}$) $\times$($M_{Scheduler} \rightarrow P_{Object}$). The design space $S_{AdaptObjectScheduler}$ consists again of the three dimensions *Sch*, *Str* and *PFD*. Hereby, however, each of these sub-concepts is mapped to a property in P_{Adapt} and a property in P_{Object}. An alternative is then defined as a set of 3 tuples each of them consisting of three elements. An example of such an alternative from this design space is *((Sch, ADc, CL), (Str,ADc, OP), (PFD,ADr, CL))*. The number of the theoretically possible design alternatives that can be selected from this design space is the $(3^3)^3 = 19683$ alternatives. Obviously, it is not feasible anymore to depict all the alternatives explicitly.

5.3.3 Reducing Design Spaces

In principle, it is possible to list all the alternatives and analyze and select them separately. However, for large design spaces the number of alternatives soon may lead to a combinatorial explosion and likewise the identification and reasoning about individual alternatives may become very difficult. Moreover, not all the alternatives may be feasible or possible at all and it would be worthwhile to eliminate these alternatives from the design space so that the software engineer does not need to take them into account.

The following techniques can be used to reduce the design space:

1. *Selection of a sub-space*, whereby the software engineer chooses a set of alternatives from the design space.
2. *Elimination of a sub-space*, whereby non-feasible alternatives are ruled out based on the constraints as derived from the solution domain or the client requirements.
3. *Heuristics-Based Selection and/or Elimination*, whereby the selection and/or elimination of the alternatives are supported by heuristic rules of methods.

The decision for reducing the design space may depend on the number of alternatives that can be selected from the given design space. Figure 5.7 illustrates the flow diagram that represents the strategy for design space reduction.

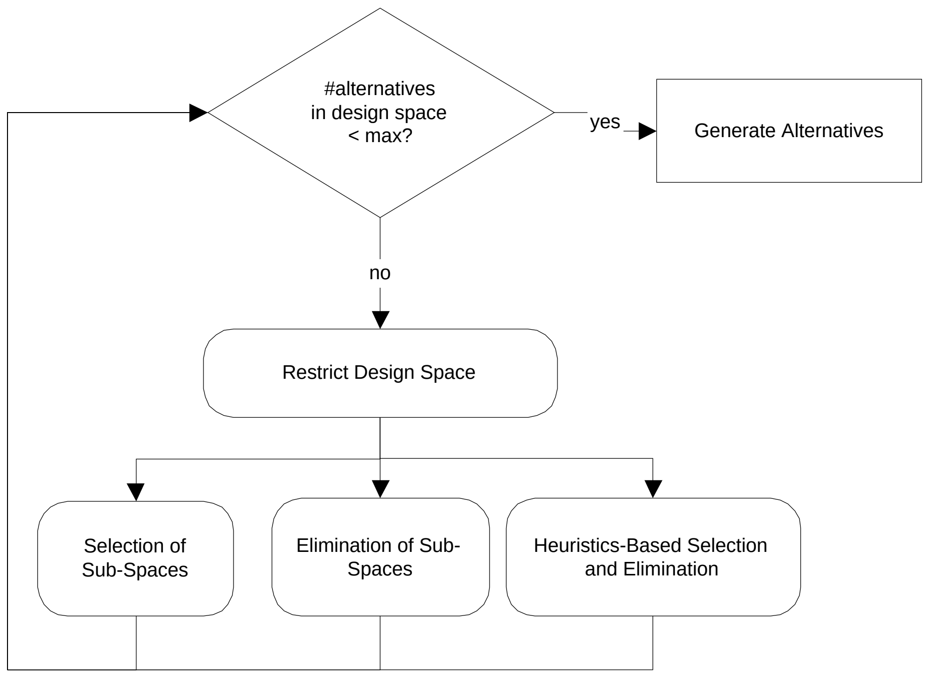

#alternatives in design space < max?
yes
Generate Alternatives
no
Restrict Design Space
Selection of Sub-Spaces
Elimination of Sub-Spaces
Heuristics-Based Selection and Elimination

Figure 5.7 *Strategy for design space reduction*

For a given design space the software engineer may compute the number of alternatives. In case it is less than a pre-defined maximum the software engineer may decide to list the alternatives and analyze them one by one; otherwise the design space may be reduced using the previous techniques. If the number of alternatives is still larger than the pre-defined maximum then another reduction step may be performed. This process will continue until the number of alternatives is less than or equal to the maximum. In the following sub-sections we will focus on each reduction technique in more detail.

Selection of a Sub-Space

We distinguish three possible techniques for the selection of a sub-space of a design space:

1. Direct selection from the design space.
2. Selection based on conditional specifications
3. Matrix-based selection

We will explain these techniques in the following.

Direct Selection

This is a simple technique in which the software engineer directly selects the required alternatives from the design space. For this the total list of the alternatives may be shown and scanned by the software engineer, or the software engineer may just directly indicate his/her wishes without giving any notice to the list of the alternatives. Consider for example the design space $S_{ObjectScheduler}$. The first option can then be visualized as follows:

$$
\begin{gathered}
S_{ObjectScheduler} = \\
\{ \boxed{((Sch,CL)\ (Str,CL)\ (PFD,CL))},\ \boxed{((Sch,CL)\ (Str,CL)\ (PFD,OP))},\quad ((Sch,CL)\ (Str,CL)\ (PFD,AT)), \\
((Sch,CL)\ (Str,OP)\ (PFD,CL)),\quad ((Sch,CL)\ (Str,OP)\ (PFD,OP)),\quad ((Sch,CL)\ (Str,OP)\ (PFD,AT)), \\
((Sch,CL)\ (Str,AT)\ (PFD,CL)),\quad \boxed{((Sch,CL)\ (Str,AT)\ (PFD,OP))},\quad ((Sch,CL)\ (Str,AT)\ (PFD,AT)), \\
((Sch,OP)\ (Str,CL)\ (PFD,CL)),\quad ((Sch,OP)\ (Str,CL)\ (PFD,OP)),\quad ((Sch,OP)\ (Str,CL)\ (PFD,AT)), \\
((Sch,OP)\ (Str,OP)\ (PFD,CL)),\quad ((Sch,OP)\ (Str,OP)\ (PFD,OP)),\quad ((Sch,OP)\ (Str,OP)\ (PFD,AT)), \\
((Sch,OP)\ (Str,AT)\ (PFD,CL)),\quad ((Sch,OP)\ (Str,AT)\ (PFD,OP)),\quad \boxed{((Sch,OP)\ (Str,AT)\ (PFD,AT))}, \\
((Sch,AT)\ (Str,AT)\ (PFD,CL)),\quad ((Sch,AT)\ (Str,AT)\ (PFD,OP)),\quad ((Sch,AT)\ (Str,AT)\ (PFD,AT)) \\
((Sch,AT)\ (Str,OP)\ (PFD,CL)),\quad ((Sch,AT)\ (Str,OP)\ (PFD,OP)),\quad ((Sch,AT)\ (Str,AT)\ (PFD,AT)) \\
\boxed{((Sch,AT)\ (Str,AT)\ (PFD,CL))},\quad ((Sch,AT)\ (Str,AT)\ (PFD,OP)),\quad ((Sch,AT)\ (Str,AT)\ (PFD,AT))\ \}
\end{gathered}
$$

Hereby the bordered alternatives represent the alternatives that the software engineer has selected. In the above list the software engineer has selected 5 alternatives.

Selection based on conditional specifications

For large design spaces, listing all the alternatives and scanning these alternatives may be a time-consuming process and as such be ruled out as a viable option. A better option in that case may be to provide a condition that specifies the set of alternatives the software engineer is interested in. Formally, the selection of alternatives from design spaces can be considered as a function space that is restricted through conditions. We can specify this in the following general form:

$$DesignSpace :: \{ concept \rightarrow propertySet \mid (condition) \}$$

Where *condition* can be made up of several functions and have one of the following forms:

- $subconcept \mapsto property$, where $subconcept \in concept$ and $property \in propertySet$.
- $condition1 \wedge condition2$, that evaluates to true if both *condition1* and *condition2* evaluate to true
- $condition1 \vee condition2$, that evaluates to true if either *condition1* or *condition2* or both of them evaluate to true.
- $\neg condition$, that is true if *condition* is false; it is false if *condition* is true.

- $\forall x(\mathit{condition})$, that evaluates to true if the condition evaluates to true for every x$\in$ *(concept* $\times$ *propertySet).*
- $\exists x(\mathit{condition})$ that evaluates to true if the condition evaluates to true for at least one x $\in$ *(concept* $\times$ *propertySet).*

The function results in a reduced design space that includes a reduced set of alternatives that meets the specified condition.

Let us now consider some examples to clarify the above ideas. Consider, for instance, the design space $S_{ObjectScheduler}$ that has been described before. To select all the alternatives from this design space in which the synchronization scheme is a class we can define a new space $S_{RobjectScheduler}$ that is a reduced sub-space of $S_{ObjectScheduler}$:

$$S_{RObjectScheduler} :: \{ M_{Scheduler} \rightarrow P_{Object} \mid (Sch \mapsto CL) \vee (Sch \mapsto OP) \}$$

This space selects the alternatives in which the synchronization scheme (SCH) has been selected as a class (CL) or an operation (OP). This results in the following 18 alternatives:

{ ((Sch,CL) (Str,CL) (PFD,CL)),   ((Sch,CL) (Str,CL) (PFD,OP)),   ((Sch,CL) (Str,CL) (PFD,AT)),
((Sch,CL) (Str,OP) (PFD,CL)),   ((Sch,CL) (Str,OP) (PFD,OP)),   ((Sch,CL) (Str,OP) (PFD,AT)),
((Sch,CL) (Str,AT) (PFD,CL)),   ((Sch,CL) (Str,AT) (PFD,OP)),   ((Sch,CL) (Str,AT) (PFD,AT)),
((Sch,OP) (Str,CL) (PFD,CL)),   ((Sch,OP) (Str,CL) (PFD,OP)),   ((Sch,OP) (Str,CL) (PFD,AT)),
((Sch,OP) (Str,OP) (PFD,CL)),   ((Sch,OP) (Str,OP) (PFD,OP)),   ((Sch,OP) (Str,OP) (PFD,AT)),
((Sch,OP) (Str,AT) (PFD,CL)),   ((Sch,OP) (Str,AT) (PFD,OP)),   ((Sch,OP) (Str,AT) (PFD,AT)) }

Figure 5.8 represents the design space after this reduction step. In this figure the shaded area represent the reduced design space and consists of 18 points that represent the above alternatives.

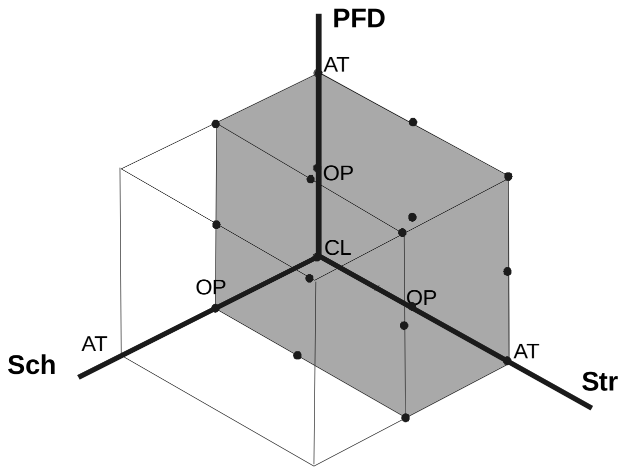


Figure 5.8 *Reduction of the design space* $S_{ObjectScheduler}$

The number of alternatives may be computed in advance so that in case this number is too large, the design space may be further reduced by providing more selection criteria. Assume that the software engineer is further interested in selecting only the alternatives in which the synchronization strategy is represented as either a class or an operation. For this the following expression needs to be specified:

$$S_{ObjectScheduler} :: \{ M_{Scheduler} \rightarrow P_{Object} \mid ((Sch \mapsto CL) \vee (Sch \mapsto OP)) \wedge ((Str \mapsto CL) \vee (Str \mapsto OP))\}$$

Note that this now results in the following 12 alternatives:

$$\begin{gathered}\{ ((Sch,CL)\ (Str,CL)\ (PFD,CL)),\quad ((Sch,CL)\ (Str,CL)\ (PFD,OP)),\quad ((Sch,CL)\ (Str,CL)\ (PFD,AT)),\\ ((Sch,CL)\ (Str,OP)\ (PFD,CL)),\quad ((Sch,CL)\ (Str,OP)\ (PFD,OP)),\quad ((Sch,CL)\ (Str,OP)\ (PFD,AT)),\\ ((Sch,OP)\ (Str,CL)\ (PFD,CL)),\quad ((Sch,OP)\ (Str,CL)\ (PFD,OP)),\quad ((Sch,OP)\ (Str,CL)\ (PFD,AT)),\\ ((Sch,OP)\ (Str,OP)\ (PFD,CL)),\quad ((Sch,OP)\ (Str,OP)\ (PFD,OP)),\quad ((Sch,OP)\ (Str,OP)\ (PFD,AT))\ \}\end{gathered}$$

We can graphically represent this reduced design space as illustrated by the shaded area in Figure 5.9.

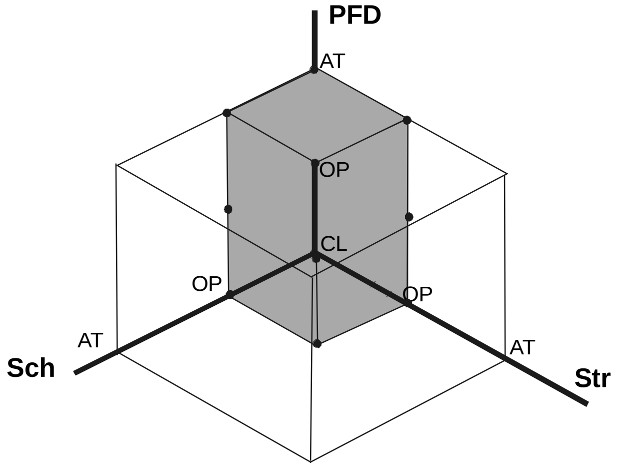


Figure 5.9 *Further reduction of the design space $S_{ObjectScheduler}$*

This selection mechanism may be very effective in reducing very large design spaces. Consider, for example, the design space $S_{AdaptObjectScheduler}$ that includes 19683 theoretically possible design alternatives. We may reduce this design space by providing, for example, the following function.

$$\begin{aligned}S_{AdaptObjectScheduler} :: \{adapt :: M_{Scheduler} \rightarrow P_{Adapt}; objectify :: M_{Scheduler} \rightarrow P_{Object} \mid \\ adapt : (Sch \mapsto ADr) \wedge (Str \mapsto ADr) \wedge (Str \mapsto ADr);\\ objectify : (Sch \mapsto CL) \wedge ((Str \mapsto CL) \vee (Str \mapsto OP))\}\end{aligned}$$

This function consists of two functions *adapt* and *objectify*. Function *adapt* defines the condition for the adaptability properties of the sub-concepts of *Scheduler* and states that the synchronization scheme (*Sch*), synchronization strategy (*Str*) and the performance failure detector (*PFD*) all need to be run-time adaptable. Function *objectify* defines the condition for the mapping to object properties and states that the synchronization scheme must be represented as a class and the synchronization strategy as either a class or an operation. The total expression reduces the design space tremendously and results only in the following 6 alternatives.

$$\{ ((Sch,ADr,CL)\ (Str,ADr,CL)\ (PFD,ADr,CL)),\ ((Sch,ADr,CL)\ (Str,ADr,OP)\ (PFD,ADr,CL)),$$
$$((Sch,ADr,CL)\ (Str,ADr,CL)\ (PFD,ADr,OP)),\ ((Sch,ADr,CL)\ (Str,ADr,OP)\ (PFD,ADr,OP)),$$
$$((Sch,ADr,CL)\ (Str,ADr,CL)\ (PFD,ADr,AT)),\ ((Sch,ADr,CL)\ (Str,ADr,OP)\ (PFD,ADr,AT)) \}$$

Matrix-based Selection

The third option in selecting sub-spaces from a design space is the matrix-based selection. Hereby, we do not list all the alternatives of the design space but rather list all the elements from which the alternatives in the design space are constituted. This list is provided by the Cartesian product of the dimensions of the design space. For example, for $S_{ObjectScheduler}$ the Cartesian product is defined as follows:

$$C_{ObjectScheduler} = M_{Scheduler} \times P_{Object}$$

$$= (Sch,Str,PFD) \times (CL,OP,AT)$$

$$= \begin{bmatrix} (Sch,CL) & (Str,OP) & (PFD,AT) \\ (Sch,CL) & (Str,OP) & (PFD,AT) \\ (Sch,CL) & (Str,OP) & (PFD,AT) \end{bmatrix}$$

The size of the elements in the Cartesian product of the sets *propertySet* and *Concept* can be defined in the following general form:

$$size(S_{propertyDomain}) = size(P_{propertySet}) \times size(Concept) = m \times n$$

The matrix defined by $C_{ObjectScheduler}$ has 9 tuples that form the basic ingredients for the 27 alternatives in $S_{ObjectScheduler}$. An alternative can be composed of this matrix by selecting one tuple from each of the three columns. The software engineer can specify the selections using logical connectives. For example, the software engineer may select the tuples *(Sch,CL)* and *(Str,OP)* which means that all the alternatives including these two tuples need to be generated. In this case this will result in three alternatives, namely:

$$\{ ((Sch,CL), (Str,OP),(PFD,CL)), ((Sch,CL), (Str,OP),(PFD,OP)), ((Sch,CL), (Str,OP),(PFD,AT)) \}$$

The generation of the alternatives for a given design space with the dimensions *propertySet* and *concept* is carried out by a generic and recursive operation *generat*e that may be specified as follows:

```
1.  procedure generate(alternativeList: array; alternative: array; depth: int)
    var i: int;
2.  begin
3.    if (depth>size(propertySet) and accept(alternative)
4.    then add(alternativeList,alternative)
5.    else for i:=1 to size(concept) do
6.            if correct(alternative,depth,propertySet[i])
7.            then alternative[depth]:=propertySet[i];  generate(depth+1)
8.            end
9.    end
10. end
```

Figure 5.10 *The generate operation for producing alternatives from a design space*

The procedure *generate* produces the alternatives by generating a tree from *propertySet* and *concept*. Thereby, leafs of the tree represent the generated alternatives. The variable *alternativeList* is used for storing the alternatives that are generated during the execution of the procedure. The variable *alternative* includes the elements that have been selected so far. The first part of the condition in line 3 checks whether *alternative* has the necessary size of elements. The function *accept* checks whether the alternative fulfills the condition of the software engineer. The function *correct* in line 6 checks whether the alternative is valid and does not include, for example, two same sub-concepts.

Elimination of a Sub-Space

The elimination of alternatives from a design space can be performed by using the techniques described in the previous selection. In this section we focus on the conditional elimination technique. Alternatives can be eliminated from a design space by specifying a selection function whereby the selection condition is negated.

For example, for the design space $S_{ObjectScheduler}$ it may be decided to eliminate all the alternatives in which a synchronization scheme (Sch) and synchronization strategy (Str) are defined as attributes.

$$S_{ObjectScheduler} :: \{ M_{Scheduler} \rightarrow P_{Object} \mid \neg((Sch \mapsto AT) \wedge (Str \mapsto AT)) \}$$

This elimination operation results in the following 12 alternatives:

(((Sch,CL) (Str,CL) (PFD,CL)), ((Sch,CL) (Str,CL) (PFD,OP)), ((Sch,CL) (Str,CL) (PFD,AT)),
((Sch,CL) (Str,OP) (PFD,CL)), ((Sch,CL) (Str,OP) (PFD,OP)), ((Sch,CL) (Str,OP) (PFD,AT)),
((Sch,OP) (Str,CL) (PFD,CL)), ((Sch,OP) (Str,CL) (PFD,OP)), ((Sch,OP) (Str,CL) (PFD,AT)),
((Sch,OP) (Str,OP) (PFD,CL)), ((Sch,OP) (Str,OP) (PFD,OP)), ((Sch,OP) (Str,OP) (PFD,AT)))

The selection and elimination operations may be utilized together to reduce the design space. For example, we may extend the previous expression by a selection operation that selects the alternatives from the design space $S_{ObjectScheduler}$ in which the performance failure detector (PFD) is represented as an operation.

$$S_{ObjectScheduler} :: \{ M_{Scheduler} \rightarrow P_{Object} \mid (\neg((Sch \mapsto AT) \wedge (Str \mapsto AT)) \wedge (PFD \mapsto OP)) \}$$

This results in the following list of 4 alternatives:

(((Sch,CL) (Str,CL) (PFD,OP)), ((Sch,CL) (Str,OP) (PFD,OP)),
((Sch,OP) (Str,CL) (PFD,OP)) ((Sch,OP) (Str,OP) (PFD,OP)))

The result of the above function is illustrated in Figure 5.11. Here, the dotted rectangle represents the space of alternatives that has been selected as a result of the function $(PFD \mapsto OP)$. The light shaded area represents the space of alternatives that has been eliminated as a result of the condition $\neg((Sch \mapsto AT) \wedge (Str \mapsto AT))$. The dark shaded and small rectangle represents the alternative space that is the total result of this expression and this consists of four points representing the above four alternatives.

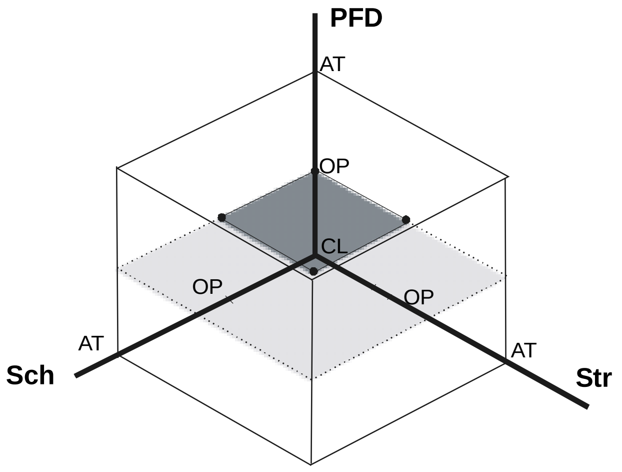


Figure 5.11 *Reduction of the design space $S_{ObjectScheduler}$ through selection and elimination of alternatives.*

Heuristics-Supported Selection and Elimination

The reduction of the design space may be supported by the utilization of heuristic rules [Riel 96]. It is not the scope of this chapter to introduce heuristic rules, rather we apply the heuristic rules as they are published in the corresponding solution domain that the dimension represents. For example, design spaces including the dimension M_{Object} we may utilize the heuristic rules from the object-oriented analysis and design methods [Jacobson et al. 99][Rumbaugh et al. 91][Wirfs-Brock et al. 90] for deciding whether an entity has to be selected as a class, operation or as an attribute. The software engineers may adopt the heuristic rules of methods that they find most appropriate, and use it to reduce the design spaces. Most methods define rules in an informal manner. Nevertheless, method rules can be expressed using conditional statements in the form **IF** <condition> **THEN** <consequent> [Tekinerdogan & Aksit 99a]. The consequent part may be an identification or elimination action and as such heuristic rules may be applied both to support the selection and the elimination operations of the reduction of the design spaces.

Consider, for instance, the design space $S_{ObjectDomain}$ from EQ5. To select alternatives from this space, we may utilize the following heuristic rules that we adapted from the method OMT [Rumbaugh et al. 91]:

> ***IF*** *an entity is relevant*
> ***THEN*** *select the entity as a class (CL)*
>
> ***IF*** *an entity describes a structural action or behavior of an object*
> ***THEN*** *select entity as an operation (OP)*
>
> ***IF*** *an entity describes another entity*
> ***THEN*** *select entity as an attribute (AT)*

Heuristic rules may also be used to eliminate alternatives from a design space [Rumbaugh et al. 91].

> ***IF*** *an entity represents a role*
> ***THEN*** *eliminate it as a class (CL)*
>
> ***IF*** *an entity is a transient event*
> ***THEN*** *eliminate it as an operation (OP)*
>
> ***IF*** *an entity is independent*
> ***THEN*** *eliminate it as an attribute (AT)*

Note that these are only examples of heuristic rules and many more rules may be extracted from the corresponding methods [Tekinerdogan & Aksit 99a]. The software engineer can apply these heuristics, provide a decision and describe these into queries. For example, using these heuristic rules it may be decided whether the synchronization scheme (Sch) should be a class, an operation or an attribute. If it is decided that it needs to be mapped to a class then the condition of the query will include the function $(SCH \mapsto CL)$. The heuristic rule application needs thus to be performed before the actual selection or elimination operations.

5.3.4 Simultaneous and Gradual Design Space Composition

Composing design spaces from existing dimensions may be carried out simultaneously or gradually [Tekinerdogan & Aksit 99b]. In simultaneous design space composition, a Cartesian product of all the dimensions is simultaneously defined and followed by a reduction and selection process. An example of this process is the construction of $S_{ObjectAdaptScheduler}$, which has been defined in section 5.3.3 in the subsection on selecting sub-spaces. The problem with the simultaneous composition approach is that dealing with too many dimensions may increase the complexity of design space reduction and alternative selection process. Moreover, not all the alternatives in this large design space may be possible or useful.

Therefore, for the software engineer it may be more practical and easier to compose design spaces gradually. Thereby not one large design space is composed but rather a sequence of design space composition and reduction processes is applied whereby each time the software engineer focuses on selected concerns. This means that the design process will include much more smaller design spaces than simultaneous composition. To combine these design spaces we use join functions. A general specification for joining two spaces is represented as follows:

$$\begin{gathered} DesignSpace :: \{ (S1, S2) \rightarrow Set \mid \\ (c, p_1), (c, p_2) \mapsto (c, (p_1, p_2)), \\ where\, (c, p_1) \in S1, (c, p_2) \in S2 \} \end{gathered}$$

Here, *S1* and *S2* represent design spaces that contain tuples with shared concepts with different properties. Consider for example the joining of the reduced spaces of $S_{AdaptScheduler}$ and $S_{ObjectScheduler}$ that are defined as follows:

$$S_{AdaptScheduler} = \{ ((Sch,ADr)\ (Str,ADr)\ (PFD,ADr)),\ \ ((Sch,ADc)\ (Str,ADr)\ (PFD,FX)) \}$$

$$S_{ObjectScheduler} = \{ ((Sch,CL)\ (Str,CL)\ (PFD,CL)),\ ((Sch,CL)\ (Str,OP)\ (PFD,OP)),\ ((Sch,CL)\ (Str,CL)\ (PFD,AT)) \}$$

The joined design space $S_{ObjectAdaptScheduler}$ may then be defined as follows:

$$\begin{gathered} S_{AdaptObjectScheduler} = \\ \{ ((Sch,CL,ADr)\ (Str,CL,ADr)\ (PFD,CL,ADr)), ((Sch,CL,ADc)\ (Str,CL,ADc)\ (PFD,CL,ADc)) \\ ((Sch,CL,ADr)\ (Str,OP,ADr)\ (PFD,OP,ADr)), ((Sch,CL,ADc)\ (Str,OP,ADr)\ (PFD,OP,FX)), \\ ((Sch,CL,ADr)\ (Str,CL,ADr)\ (PFD,AT,ADr)), ((Sch,CL,ADc)\ (Str,CL,ADc)\ (PFD,AT,FX)) \} \end{gathered}$$

5.3.5 Quantifying Design Alternatives

In design algebra, it is possible to assign priority values to the tuples of the alternatives in the design space to analyze and compare the various design alternatives. The quantification of alternatives may be relevant because of two reasons. Firstly, after the reduction of the design space there may still be many alternatives left and the software engineer may need to balance between the various alternatives. Assigning quantification values may provide an ordering between the various alternatives. Secondly, the quantification of the alternatives may be used in the expressions for reducing the design spaces using selection or elimination operations.

The form of a quantification function is as follows:

$$DesignSpace :: \{ concept \times propertySet) \rightarrow N$$

The quantification function maps a tuple *(c_i, p_i)* whereby $c_i \in$ concept and $p_j \in$ propertySet, to a natural number. The function can also include various arithmetic operations such as assignment (denoted by :=), inc(x), dec(x), times, divide, or a more complex arithmetic function *function(x)*.

Consider for instance the design space $S_{AdaptScheduler}$ that includes the dimensions *Adapt* and *Scheduler* and consists of 9 tuples from which 27 alternatives may be derived. For the example problem it may be the case that for *Scheduler* the run-time adaptability of the synchronization scheme (Sch) is considered as absolutely important. The concepts synchronization strategy (Str) and performance failure detector (PFD) are considered less vital and they may also be compile-time adaptable but not fixed. These requirements can be expressed by assigning priority values to the individual tuples in the space $S_{AdaptScheduler}$. The following formula represents a specification that provides an example of the assignment of various priority values to the sub-concepts of $S_{AdaptScheduler}$:

$$\begin{aligned}
S_{AdaptScheduler} :: \{ (M_{Scheduler} \times P_{Adapt}) \rightarrow N) | \\
\forall (c,p), c \in M_{Scheduler}, p \in P_{Adapt} : (c,p) \mapsto 0; \\
((Sch,ADr) \mapsto 10); \\
((Str,ADr) \mapsto 10) \wedge (Str,ADc) \mapsto 5)); \\
((PFD,ADr) \mapsto 10) \wedge (PFD,ADc) \mapsto 5))\}
\end{aligned}$$

Hereby the first function initializes the priorities of all the tuples to 0. The function $(Sch, ADr) \mapsto 10$ assigns to all the tuples *(Sch,ADr)* in every alternative in the design space $S_{AdaptScheduler}$ the priority value 10. Similarly, the tuples *(Str,ADr)* and *(Str, ADc)* are assigned respectively the priority values 10 and 5 to denote that for the concept synchronization strategy the run-time adaptability has a higher preference over compile-time adaptability. The same priorities of the sub-concept *Str* are assigned to the sub-concept *PFD*. The result of this quantification function is that all the 27 alternatives of the design space $S_{AdaptScheduler}$ will have been assigned a priority value. For illustration purposes, Table 5.6 lists 5 of these 27 alternatives.

Alternative no.	Design Alternative	Priority degree
1.	(((Sch,ADr), 10) ((Str,ADr),10) ((PFD,ADr),10))	30
2.	(((Sch,ADr), 10) ((Str,ADc), 5) ((PFD,FX),0))	15
3.	(((Sch,ADc), 0) ((Str,ADc), 5) ((PFD,FX),0))	5
4.	(((Sch,ADc), 0) ((Str,ADc),5) ((PFD,ADc),5))	10
5.	(((Sch,FX), 0) ((Str,FX), 0) ((PFD,FX),0)	0

Table 5.6 *Set of quantified design alternatives from the design space* $S_{AdaptScheduler}$.

The column Priority Degree in this table represents the value of the sum of the individual tuple elements. For example, in the first alternative all the sub-concepts of *Scheduler* are defined as run-time adaptable (ADr) and therefore they have been assigned the priority degree 10. The sum of these three tuple elements is 30 as it can be seen in the table. The fifth alternative has a priority degree of 0 because all the sub-concepts of *Scheduler* have been selected as fixed (FX).

5.3.6 Formalizing Couplings between Concepts

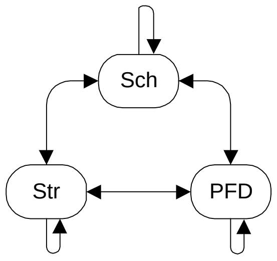
Usually, the sub-concepts of a concept are not all distinct but somehow they are connected to each other. To reason about these couplings it is required that we formalize these in an appropriate manner. In design algebra, the Cartesian product is used to represent the couplings between sub-concepts.

We define a *coupling space* of *concept1* and *concept2* as the set of the possible couplings between all the sub-concepts of *concept1* and *concept2*. The coupling space may be defined among two independent concepts but in addition it may also be defined to define the coupling space of the sub-concepts within a concept. For example, all the theoretically possible couplings of the sub-concepts of $M_{Scheduler}$ may be specified as follows:

$$S_{SchedulerCoupling} :: M_{Scheduler} \times M_{Scheduler}$$

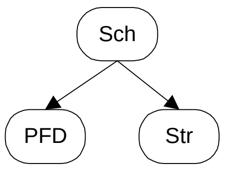
$$= (Sch, Str, PFD) \times (Sch, Str, PFD)$$

$$= \begin{bmatrix} (Sch,Sch) & (Str,Sch) & (PFD,Sch) \\ (Sch,Str) & (Str,Str) & (PFD,Str) \\ (Sch,PFD) & (Str,PFD) & (PFD,PFD) \end{bmatrix}$$



On the right of the matrix, its equivalent graph notation has been given whereby the nodes represent the sub-concepts of $M_{Scheduler}$. In the matrix, for example, the tuple *(Sch,Str)* represents the coupling between the synchronization scheme concept and the synchronization strategy concept. In practice, naturally not all of the couplings may be possible and the coupling space may generally be simplified. Usually, the solution domain and the client requirements provide the constraints for the valid couplings. For example, if we consider the conceptual model for *Scheduler* as it has been presented in Figure 5.1, we can directly derive the structural connections between the three sub-concepts and reduce the coupling space $S_{SchedulerCoupling}$ resulting in the following matrix:

$$S_{SchedulerCoupling} = \begin{bmatrix} - & - & - \\ (Sch, Str) & - & - \\ (Sch, PFD) & - & - \end{bmatrix}$$



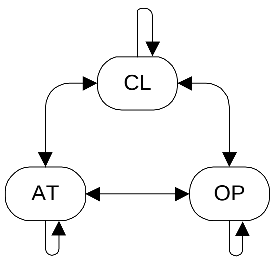
The symbol '-' in the matrix denotes that the corresponding relation is not valid. The above matrix defines two couplings namely *(Sch,Str)* and *(Sch,PFD)*. On the right of the matrix the graphical representation of the couplings between the concepts has been given.

The couplings of the sub-concepts of solution domain concept are more or less defined by the solution domain. The realization of the couplings between the sub-concepts in the object-oriented domain can be done in many different ways. Let us explain this a little more. To represent all the possible couplings between the elements *CL*, *OP* and *AT* of P_{Object}, the following coupling space $S_{ObjectCoupling}$ is defined:

$$S_{ObjectCoupling} :: P_{Object} \times P_{Object}$$

$$= (CL, OP, AT) \times (CL, OP, AT)$$

$$= \begin{bmatrix} (CL,CL) & (OP,CL) & (AT,CL) \\ (CL,OP) & (OP,OP) & (AT,OP) \\ (CL,AT) & (OP,AT) & (AT,AT) \end{bmatrix}$$



For example the tuple *(CL, CL)* represents the coupling between two classes. On the right of the matrix the graph notation has been given. To validate whether all these couplings exist we need to analyze the existing object-oriented relations. This process is again similar to a solution domain analysis process, whereby the solution domain is now the object-oriented model. The object-oriented model provides several relations among its concepts among which the basic three relations are the inheritance, the aggregation and the association relation. Based on this we define the following property set for the object-oriented relations:

$$P_{ObjectRelation} = (IN, AG, AS)$$

Hereby, IN, AG and AS correspond to the relations of inheritance, aggregation and association respectively. The set of all the possible relations using two concepts and a relation of the object-oriented model can now be specified by the following function.

$$S_{ObjectCoupling\,Relation} :: (P_{Object} \times P_{Object}) \rightarrow P_{Object\,Relation}$$

This function maps a couple *(c_1,c_2)* to a relation *r*, whereby c_1 and $c_2 \in P_{Object}$ and r$\in P_{ObjectRelation}$. The possible bindings of the function $P_{ObjectRelation}$ is equal to the product of the size of $S_{ObjectCoupling}$ and the size of $P_{ObjectRelation}$, namely (3x3) x 3 = 27. Table 5.7 presents all these relations whereby the relations are classified into 9 categories that each represent the three relations *IN*, *AS* and *AG* for a given combination of the elements *CL*, *OP* and *AT* of P_{Object}. In the figure UML-like notations are used to represent the object-oriented abstractions. A class is represented as a rectangle whereby the top part represents the name of the class. In the table we use the symbol *op* to describe an operation and *at* to describe an attribute. The inheritance relation is represented by a line with a hollow arrowhead pointing to the parent. The association relation is represented through directed arrows. The aggregation relation is represented by a line with a diamond head.

We can use this table to define the relations between the solution domain concepts. Assume that, for example, the following alternative is selected from $S_{ObjectScheduler}$ to realize as an object model:

Alternative = ((Sch,CL) (Str,CL) (PFD,OP))

Based on the couplings in $S_{SchedulerCoupling}$ we can define the following model:

{ ((Sch,CL),(Str,CL)), ((Sch,CL),(PFD,OP)) }

The first tuple in this model means that the sub-concept *Sch* is represented as a class and connects to *Str* that is also a class. The second tuple connects the class *Sch* with the operation *PFD*. From Table 5.7 we can derive that there are 3 possible ways of connecting two classes and 3 possible ways of connecting a class with an operation.

The software engineer needs thus to select from 3x3 + 3x3 = 18 different combinations of relations, for instance *{ (((Sch,CL),(Str,CL)), AG), (((Sch,CL),(PFD,OP)), AG) }* would be an alternative.

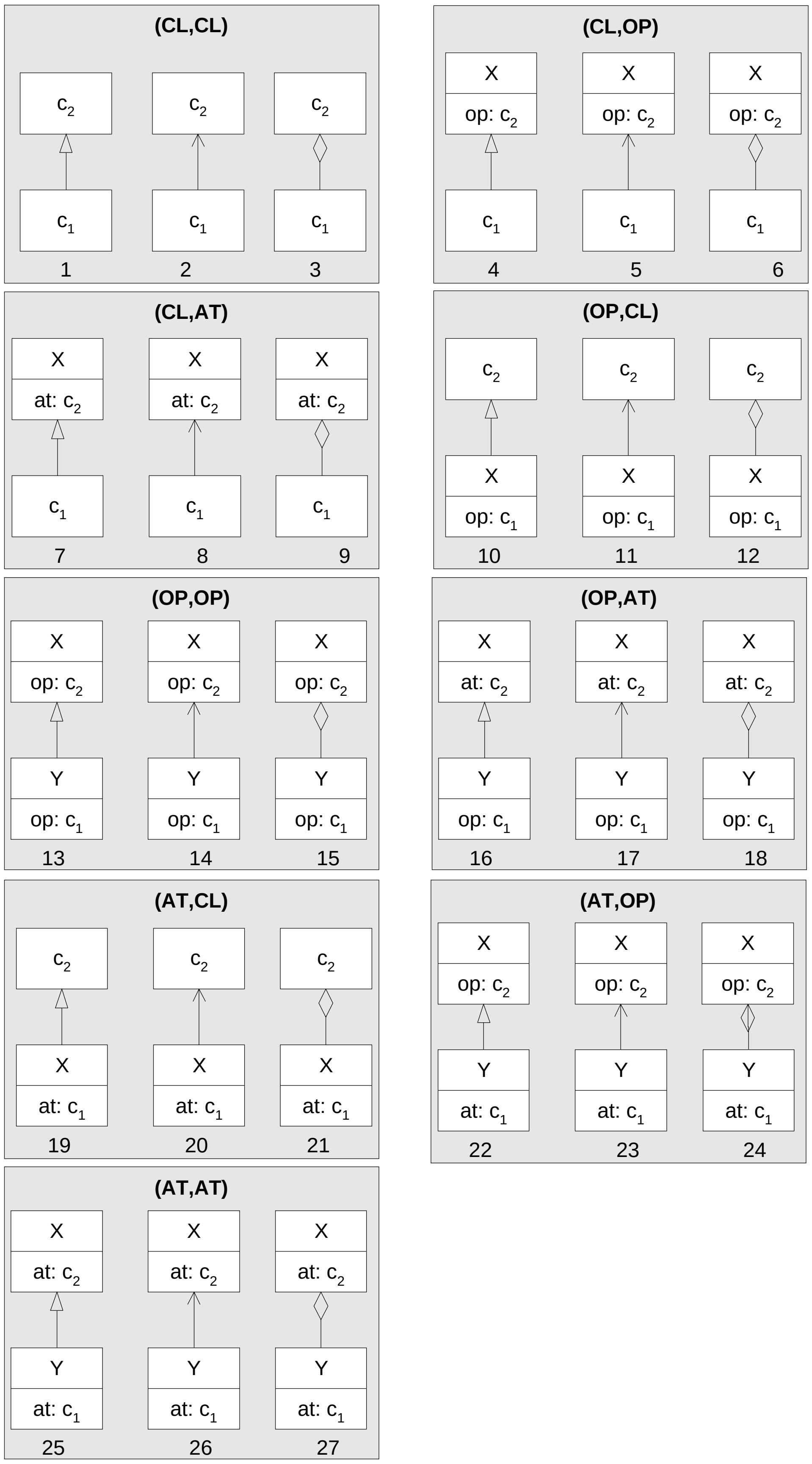


Table 5.7 *Possible relations between the concepts within the object model*

The selection process may be supported by the heuristic rules on object-oriented relations, which can be identified from various publications on object-oriented methods. Consider for instance the following heuristics for the identification of an inheritance, aggregation and association relation [Rumbaugh et al. 91]:

> ***IF*** *a concept1 is a specialization of a concept2*
> ***THEN*** *select an inheritance relation (IN)*
>
> ***IF*** *a concept1 is a part of concept2*
> ***THEN*** *select an aggregation relation (AG)*
>
> ***IF*** *a concept1 is a structural acquaintance of concept2*
> ***THEN*** *select an association relation (AS)*

Utilizing these heuristic rules the software engineer may decide that *Str* is a part of *Sch* and *Sch* has an association relation with *PFD*, resulting in the following model:

{ (((Sch,CL),(Str,CL)), AG), (((Sch,CL),(PFD,OP)), AS) }

If we inspect the possible object relations of Table 5.7 then the only possible pattern for the first tuple is the pattern with the number 1. For the second tuple we can only select the pattern with number 5. As a result of this, the object model for the above alternative can be represented as given in Figure 5.12.

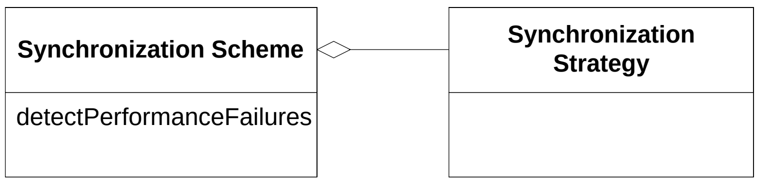


Figure 5.12 *Object model derived from the tuple { (((Sch,CL),(Str,CL)), AG), (((Sch,CL),(PFD,OP)), AS) }*

5.4 Process for Deriving Adaptable Object-Oriented Schedulers

In this section, we will introduce a process for deriving adaptable object-oriented design alternatives for the concept *Scheduler* in the transaction system architecture The objective of this process is to gradually introduce domain, design and implementation knowledge and selecting the alternatives based on the adaptability factors. This example process consists of the following phases:

1. Identification of the adaptable concepts: In this phase the software engineer decides the adaptability properties of the selected concepts. The purpose of this

phase is to make the software engineer conscious about the design decisions with respect to the adaptability characteristics of the models that (s)he develops.

2. Identification of the object-oriented abstractions: In this phase the software engineer decides the mapping of the selected alternatives to the object-oriented concepts. The result of this phase is a consciously selected set of object-oriented design alternatives.
3. Identification of the adaptable object-oriented abstractions: The concepts with the required adaptability properties delivered from the previous phase are classified according to the object-oriented abstraction techniques. The result of this phase is a consciously selected set of object-oriented abstractions with well-defined adaptability characteristics.
4. Identification of the object-oriented relations: This phase aims to identify the relations among the identified concepts. The result of this phase is a set of object-oriented relations that satisfy the adaptability requirements.

The total result of this process is a set of alternative object-oriented models, which are ordered according to their adaptability degrees. Using this ordering, the software engineer may consciously select one among them. In the following sections we will concentrate on each phase.

5.4.1 Identifying the Adaptable Concepts

To identify the adaptable concepts of Scheduler we will use the function $S_{AdaptScheduler}$ that maps $M_{Scheduler}$ to P_{Adapt}. This function defines the alternative design space that includes the following 27 theoretically possible alternatives:

$S_{AdaptScheduler} = M_{Scheduler} \rightarrow P_{Adapt} =$

{ ((Sch,ADr) (Str,ADr) (PFD,ADr)), ((Sch,ADr) (Str,ADr) (PFD,ADc)), ((Sch,ADr) (Str,ADr) (PFD,FX)),
((Sch,ADr) (Str,ADc) (PFD,ADr)), ((Sch,ADr) (Str,ADc) (PFD,ADc)), ((Sch,ADr) (Str,ADc) (PFD,FX)),
((Sch,ADr) (Str,FX) (PFD,ADr)), ((Sch,ADr) (Str,FX) (PFD,ADc)), ((Sch,ADr) (Str,FX) (PFD,FX)),
((Sch,ADc) (Str,ADr) (PFD,ADr)), ((Sch,ADc) (Str,ADr) (PFD,ADc)), ((Sch,ADc) (Str,ADr) (PFD,FX)),
((Sch,ADc) (Str,ADc) (PFD,ADr)), ((Sch,ADc) (Str,ADc) (PFD,ADc)), ((Sch,ADc) (Str,ADc) (PFD,FX)),
((Sch,ADc) (Str,FX) (PFD,ADr)), ((Sch,ADc) (Str,FX) (PFD,ADc)), ((Sch,ADc) (Str,FX) (PFD,FX)),
((Sch,FX) (Str,FX) (PFD,ADr)), ((Sch,FX) (Str,FX) (PFD,ADc)), ((Sch,FX) (Str,FX) (PFD,FX))
((Sch,FX) (Str,ADc) (PFD,ADr)), ((Sch,FX) (Str,ADc) (PFD,ADc)), ((Sch,FX) (Str,FX) (PFD,FX))
((Sch,FX) (Str,FX) (PFD,ADr)), ((Sch,FX) (Str,FX) (PFD,ADc)), ((Sch,FX) (Str,FX) (PFD,FX)) }

The software engineer may now use the techniques of direct selection, condition-based selection or matrix-based selection to select the required alternatives. Assume that as a result of this selection process the space is reduced to the following 4 alternatives:

$S_{RAdaptScheduler}$ ={ ((Sch,ADr) (Str,ADr) (PFD,ADr)), ((Sch,FX) (Str,FX) (PFD,FX)), ((Sch,ADr) (Str,ADr) (PFD,ADc)), ((Sch,ADr) (Str,FX) (PFD,FX)) }

5.4.2 Identifying the Object-Oriented Abstractions

The total alternatives of $S_{ObjectScheduler}$ have been given in section 5.3.2. Assume that the following three alternatives have been selected:

$S_{RObjectScheduler}$ = { ((Sch,CL) (Str,CL) (PFD,CL)), ((Sch,CL) (Str,OP) (PFD,OP)), ((Sch,CL) (Str,AT) (PFD,OP)) }

In this reduced space the software engineer has selected the synchronization scheme as a class.

5.4.3 Identifying the Adaptable Object-Oriented Concepts

For identifying the adaptable object-oriented concepts we need to join the spaces $S_{RadaptScheduler}$ and $S_{RobjectScheduler}$. This results in the space $S_{JoinedRAdaptRObjectScheduler}$ that consists of 12 alternatives:

$S_{JoinedRAdaptRObjectScheduler}$=
{ ((Sch,ADr,CL) (Str,ADr,CL) (PFD,ADr,CL)), ((Sch,ADr,CL) (Str,ADr,OP) (PFD,ADr,OP)),
((Sch,ADr,CL) (Str,ADr,AT) (PFD,ADr,OP)), ((Sch,FX,CL) (Str,FX,CL) (PFD,FX,CL)),
((Sch,FX,CL) (Str,FX,OP) (PFD,FX,OP)), ((Sch,FX,CL) (Str,FX,AT) (PFD,FX,OP)),
((Sch,ADr,CL) (Str,ADr,CL) (PFD,ADc,CL)), ((Sch,ADr,CL) (Str,ADr,OP) (PFD,ADc,OP)),
((Sch,ADr,CL) (Str,ADr,AT) (PFD,ADc,OP)), ((Sch,ADr,CL) (Str,FX,CL) (PFD,FX,CL)),
((Sch,ADr,CL) (Str,ADr,OP) (PFD,FX,OP)), ((Sch,ADr,CL) (Str,FX,AT) (PFD,FX,OP)) }

We may further reduce this space by considering the constraints on the relations between the object and the adaptability properties. Assume that the software engineer applies the constraint that operations in the object-model cannot be run-time adaptable, then the $S_{JoinedRAdaptRObjectScheduler}$ will result in a reduced space:

$S_{R\_JoinedRAdaptRObjectScheduler}$=
{ ((Sch,ADr,CL) (Str,ADr,CL) (PFD,ADr,CL)), ((Sch,FX,CL) (Str,FX,CL) (PFD,FX,CL)),
((Sch,FX,CL) (Str,FX,OP) (PFD,FX,OP)), ((Sch,FX,CL) (Str,FX,AT) (PFD,FX,OP)),
((Sch,ADr,CL) (Str,ADr,CL) (PFD,ADc,CL)), ((Sch,ADr,CL) (Str,ADr,AT) (PFD,ADc,OP)),
((Sch,ADr,CL) (Str,FX,CL) (PFD,FX,CL)), ((Sch,ADr,CL) (Str,FX,AT) (PFD,FX,OP)) }

5.4.4 Identification of the Object-Oriented Relations

The identification of the object-oriented relations is carried out by first considering the couplings for each alternative and then by mapping these to the object-oriented relations. We explain these two sub-steps separately.

1. *Define the coupling between the concepts for each alternative*
   As described in section, 5.3.6, only the couplings *(Sch,Str)* and *(Sch,PFD)* are possible. For example, the first alternative *((Sch,ADr,CL) (Str,ADr,CL) (PFD,ADr,CL))* of $S_{R\_JoinedRAdaptRObjectScheduler}$ will be mapped to { *((Sch,ADr,CL),(Str,ADr,CL)), ((Sch,ADr,CL),(PFD,ADr,CL))* }. Graphically this is represented as follows:

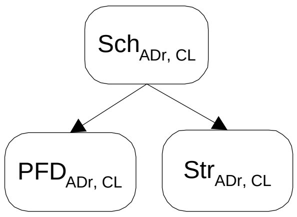


Figure 5.13 *Graphical notation for the tuple ((Sch,ADr,CL),(Str,ADr,CL)), ((Sch,ADr,CL),(PFD,ADr,CL))*

The 8 alternative models that can be derived from $S_{R\_JoinedRAdaptRObjectScheduler}$ are represented by the following space.

$S_{Coupled\_R\_JoinedRAdaptRObjectScheduler}$ =

{ *((Sch,ADr,CL),(Str,ADr,CL)) - ((Sch,ADr,CL),(PFD,ADr,CL))*
((Sch,FX,CL),(Str,FX,CL)) - ((Sch,FX,CL), (PFD,FX,CL)),
((Sch,FX,CL),(Str,FX,OP)) - ((Sch,FX,CL),(PFD,FX,OP)),
((Sch,FX,CL),(Str,FX,AT)) - ((Sch,FX,CL),(PFD,FX,OP)),
((Sch,ADr,CL),(Str,ADr,CL)) - ((Sch,ADr,CL),(PFD,ADc,CL)),
((Sch,ADr,CL),(Str,ADr,AT)) - ((Sch,ADr,CL),(PFD,ADc,OP)),
((Sch,ADr,CL),(Str,FX,CL)) - ((Sch,ADr,CL),(PFD,FX,CL)),
((Sch,ADr,CL),(Str,FX,AT)) - ((Sch,ADr,CL)(PFD,FX,OP)) }

2. *Map the identified couplings to object-oriented couplings as defined by $P_{ObjectRelation}$*
   This step results in alternatives that have concepts and relations and as such can be can be easily mapped to an object-oriented notation. Mapping every alternative in $S_{Coupled\_R\_JoinedRAdaptRObjectScheduler}$ to $P_{ObjectRelation}$ results in 3^2 =9 models because every model has two relations that can be mapped to IN, AS, or AG. If we consider all the alternatives in $S_{Coupled\_R\_JoinedRAdaptRObjectScheduler}$ then we can thus derive 8 x 9 = 72 theoretically possible models. We may reduce this set by applying object-oriented heuristics for identifying inheritance, association and aggregation, which have been given in section 5.3.6. Assume that the software engineer decides that the synchronization strategy (Str) and the performance failure detector (PFD) are a part of the synchronization scheme (Sch). The consequence of this is that we only select the models that apply aggregation (AG) relation and this reduces the set of 72 models to only 8 models, which are listed in Table 5.8:

Model no.	Model	Priority
1.	*((Sch,ADr,CL),(Str,ADr,CL),AG) - ((Sch,ADr,CL),(PFD,ADr,CL),AG)*	30
2.	*((Sch,FX,CL),(Str,FX,CL),AG) - ((Sch,FX,CL), (PFD,FX,CL)),AG)*	0
3.	*((Sch,FX,CL),(Str,FX,OP),AG) - ((Sch,FX,CL),(PFD,FX,OP),AG)*	0
4.	*((Sch,FX,CL),(Str,FX,AT),AG) - ((Sch,FX,CL),(PFD,FX,OP),AG)*	0
5.	*((Sch,ADr,CL),(Str,ADr,CL),AG) - ((Sch,ADr,CL),(PFD,ADc,CL),AG)*	25
6.	*((Sch,ADr,CL),(Str,ADr,AT),AG) - ((Sch,ADr,CL),(PFD,ADc,OP),AG)*	25
7.	*((Sch,ADr,CL),(Str,FX,CL),AG) - ((Sch,ADr,CL),(PFD,FX,CL),AG)*	10
8.	*((Sch,ADr,CL),(Str,FX,AT),AG) - ((Sch,ADr,CL)(PFD,FX,OP),AG)*	10

Table 5.8 *The total set of object-oriented adaptable alternative models*

5.5 Designing for Time Performance

In this section we will focus on evaluating the design alternatives based on the time performance quality factor.

Assume that the software engineer selects the alternatives with model number 2 and 7 in Table 5.8.

(((Sch,ADr,CL),(Str,FX,CL)),AG) - (((Sch,ADr,CL),(PFD,FX,CL)),AG))

(((Sch,FX,CL),(Str,FX,CL)),AG) - (((Sch,FX,CL), (PFD,FX,CL)),AG))

The sub-concepts *Sch*, *Str* and *PFD* each correspond to a set of instantiations that represent the different implementation alternatives. To analyze the time-performance of both design alternatives we need to describe the various implementation alternatives to which the system can be adapted.

From the solution domain analysis in the previous chapter we derived the alternative space of the instances for each of these sub-concepts. We describe these alternative instances as properties over the sub-concepts, in a similar way as we defined the adaptability and the object properties.

For the synchronization scheme sub-concept we derive the following property set from the solution domain:

$$P_{Scheme} = (LK, TO, OPT)$$

Hereby the properties *LK*, *TO* and *OPT* represent the two-phase locking, timestamp ordering and the optimistic concurrency control schemes, respectively.

The property set for the synchronization strategy we define as follows:

$$P_{Strategy} = (AGG, CONS)$$

The properties *AGG* and *CONS* represent the aggressive and the conservative synchronization strategy, respectively.

Finally the performance failure detector is defined as follows:

$$P_{PFD} = (DL, IB, CR, IR)$$

Hereby, *DL*, *IB*, *CR* and *IR* represent the deadlock detector, infinite blocking detector, cyclic restarting detector and the infinite restarting performance failure detectors, respectively. We can depict this space graphically as it is shown in Figure 5.14.

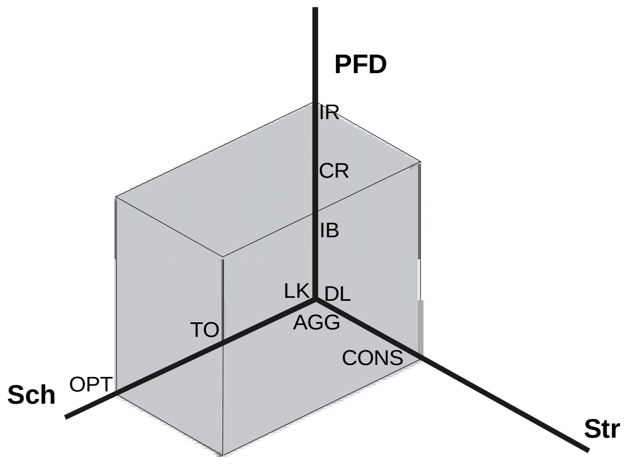


Figure 5.14 *Design space of Scheduler with instances of sub-concepts as property sets*

The product of the size of these property sets defines the theoretically possible set of alternative instantiations of *Scheduler*, that is, 3 x 3 x 4 = 36. Each of these alternatives may probably have different time-performance values. Adaptability of *Scheduler* is defined as the ability to switch between these various implementation alternatives. Let us now analyze the two alternatives, which have been selected from Table 5.8, from this perspective.

In the first alternative, the scheduler scheme is selected as run-time adaptable while the scheduling strategy and the performance failure detector are fixed. The run-time adaptability of the scheduler scheme means that the system may switch between the synchronization scheme instances, which are listed as two-phase locking (LK), timestamp-ordering (TO) and optimistic (OPT). This adaptation may help to tune to the scheduler implementation alternatives with the highest time performance for a given context.

In the selected second alternative all the tuples are fixed, which means that for each of the sub-concepts a fixed implementation alternative is chosen. Note that this can be theoretically done in 36 ways, that is, the number of alternatives. To provide a high time performance for this alternative, one may first analyze the context and then select the appropriate scheduler instances.

To analyze and measure the performance of different scheduler implementations a pilot project has been carried out [Schopbarteld 99] for which a simulation framework and a simulation environment has been developed. Hereby, the time-performance was analyzed and measured for the following situations:

1. Fixed scheduler with a two-phase locking scheme concurrency control scheme.
2. Fixed scheduler with a timestamp ordering concurrency control scheme
3. Fixed scheduler with an optimistic concurrency control scheme.
4. Run-time adaptable scheduler

Note that the first situation corresponds to the first selected alternative and the third to the fourth situations correspond to the second alternative that has been selected from Table 5.8.

For each of these four situations the time-performance was measured through a number of experiments in which the number of transactions completed per second, that is the *throughput*, has been adopted as the dependent variable[41]. The independent variable that was manipulated to provide the result of the throughput variable was the *multi-programming level*, which represents the number of transactions that may be active at a given time.

For the fixed scheduler situations an environment was set up in which the atomic transaction system only applied one scheduler implementation and the throughput was measured for the multi-programming levels between 0 and 100.

For the situation in which the run-time adaptability of the scheduler was required dynamic adaptation mechanisms have been developed. Hereby, a dynamic switching algorithm applies heuristic rules to select different alternative scheduler implementations for different contexts. The switching of the schedulers aimed to optimize the time performance, in this case the throughput, for the multi-programming levels between 0 and 100. The applied heuristic rules have been extracted from the solution domain that define the switching criteria [Kumar 96][Atkins & Coady 92][Agrawal et al. 87][Carey & Stonebraker 84] to optimize the throughput for the changing multi-programming levels. The following three rules have been basically applied:

[41] In experimental analysis and design a set of *independent variables* represents aspects that are manipulated to measure the outcome of their result on another aspect, the *dependent variable*, of the subject of domain [Fenton & Pfleeger 97].

IF *MPL.value < lowerThreshold*
THEN *select* optimistic scheduler

IF *MPL.value* $\geq$ *lowerThreshold* **and** *MPL.value* $\leq$ *upperThreshold*
THEN *select* two-phase locking scheduler

IF *MPL.value > upperThreshold*
THEN *select* timestamp-ordering scheduler

The variables *lowerThreshold* and *upperThreshold* represent the threshold values at which the selection mechanisms needs to switch to a different scheduler to increase the throughput of the system. These heuristic rules may only be effective in optimizing the time-performance if the right threshold values have been selected. These threshold values are correct if the throughput value with the selection mechanism is higher than the throughput value for a fixed implementation. To determine the initial threshold values a number of simulations have been carried out on beforehand. Figure 5.15 represents the result of these simulations [Schopbarteld 99].

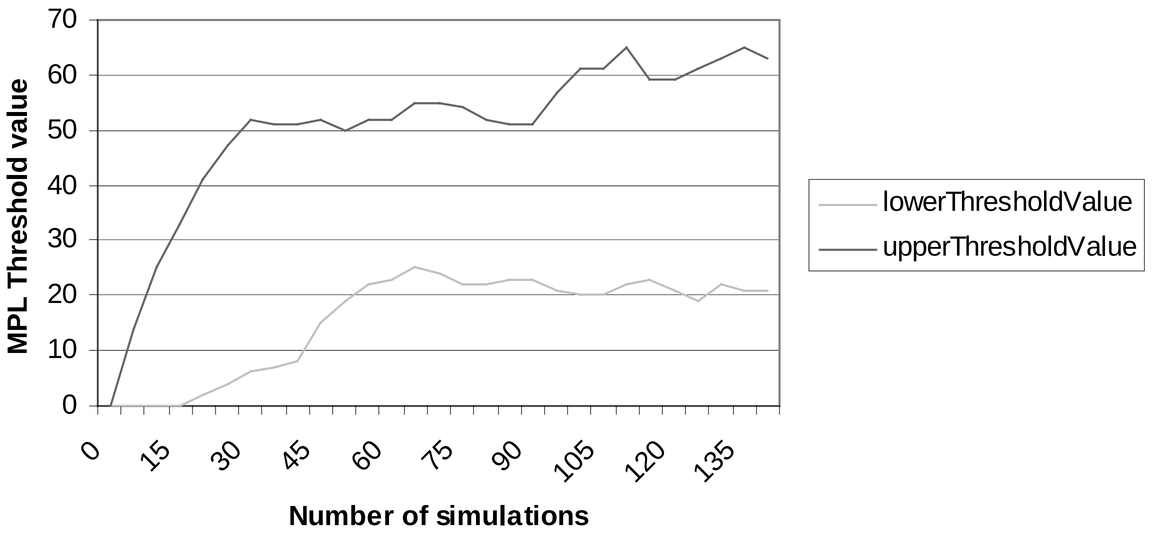


Figure 5.15 *Results of the determination of the threshold values for the MPL*

From this figure it follows that after a number of simulations the lower and upper threshold values increase to certain value and get as good as stabilized. These values have been initiated in the heuristic rules of the switching mechanism. During the actual simulations the threshold values were also dynamically adapted.

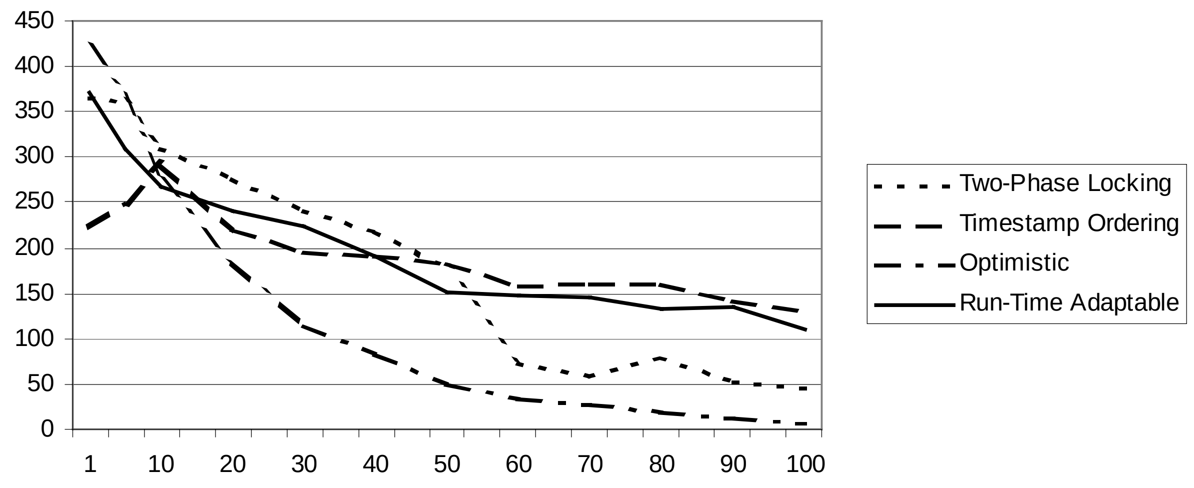


Figure 5.16 *Time-performance values for the fixed scheduler implementations and the run-time adaptable scheduler implementations.*

Figure 5.16 shows the throughput values for each scheduler implementation and the run-time adaptable selection mechanism as it has been derived from the simulations for varying multi-programming levels. From this figure we can derive that for lower multi-programming levels the optimistic scheduler implementation performs better than the timestamp ordering and the two-phase locking scheduler implementations. For higher multi-programming levels the throughput for the optimistic scheduler implementation decreases significantly and the timestamp-ordering scheduler implementation then provides a better throughput. The run-time adaptable scheduler implementation nearly follows the optimal values of the different schedulers and proofed to be effective in optimizing the time-performance.

For a further analysis of the experimental results and the details of the simulation framework and the simulation environment we refer to [Schopbarteld 99].

5.6 Automated Support for Design Algebra: *Rumi*

An additional value of design algebra can be achieved if the provided techniques are supported by a number of tools. In this section we will describe the CASE tool environment *Rumi*[42] that we have developed for supporting the design algebra techniques.

[42] Derived from the name of the famous mystic poet, Mevlana Celaleddin Rumi (1207-1273). Rumi has written extended volumes on spiritual teachings for increasing the level of human consciousness and universal human qualities such as love, generosity, patience, courage, humility and wisdom.

5.6.1 Method Engineering

The architecture of the tool environment has been developed through *method engineering* techniques. *Method engineering,* is defined as an engineering discipline for designing, constructing and adapting methods, techniques and tools for the development of information systems [Brinkkemper 96][Saeki 98][Tolvannen et. al 96]. The usually adopted method engineering process is shown in Figure 5.17. Method engineers model methods and for this they will typically analyze the *method domain*, that is, the area in which the method is described. This may include method experts, books, existing CASE tools supporting the method etc. Method engineers formally represent methods or parts of methods and store these in a so-called *method-base* for later reuse. The engineering of methods using automatic tool support is called *Computer Aided Method Engineering* (*CAME*). If needed, suitable method fragments can be retrieved from the method-base, if necessary adapted, and finally integrated into a new method. The software engineer will use CASE tools, which provide programmed method, and guides the software engineer in developing software.

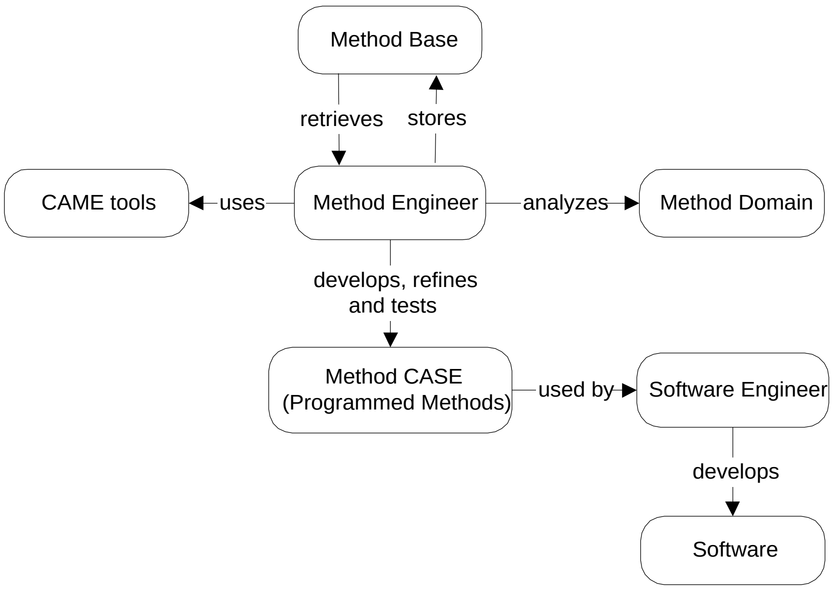


Figure 5.17 *Method Engineering Process, adapted from [Saeki 98]*

5.6.2 Meta-Model

Method engineers usually apply meta-modeling techniques to store the formally represented method in the *method-base*. The meta-models are supported by CAME tools, with which methods may be tailored.

We applied the process of Figure 5.17 for the development of the environment *Rumi* for which we applied the Parcplace Visualworks Smalltalk environment for constructing object-oriented applications. This environment can be considered as part of our CAME environment. The meta-model that has been developed in this environment and on which all the tools of *Rumi* rely is represented in *Figure 5.18*.

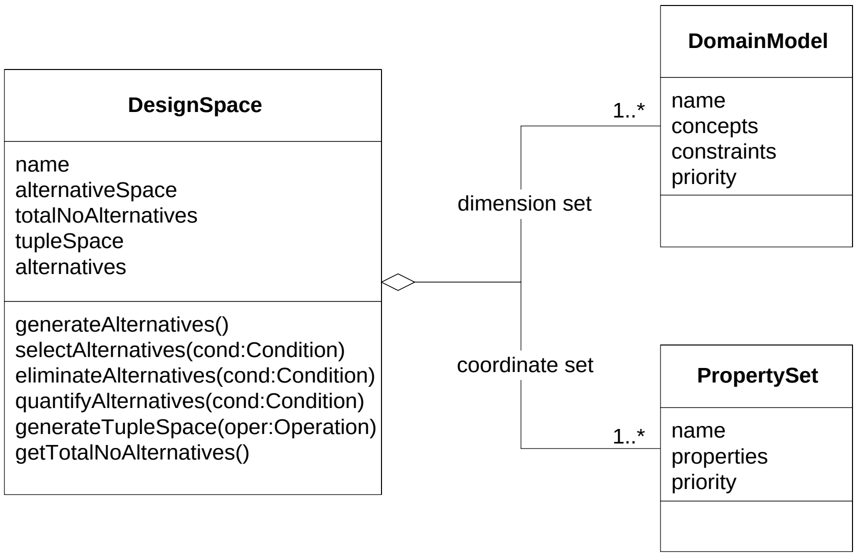


***Figure 5.18** The meta-model for the tools of Rumi*

The class *DesignSpace* represents the concept of design spaces and consists of the following attributes:

- *name*, defines a unique name for the design space.
- *dimension set*, represent the set of dimensions from which the design space will be composed. Dimensions are represented by concepts of canonical model that have been derived from the solution domain analysis.
- *coordinate set*, represents the set of properties that are used as coordinates for the dimensions of the design space. Coordinates are derived from the properties of a property set.
- *alternativeSpace*, defines the reduced set of alternative space that has resulted after reduction operations. Initially this represents the total design space.
- *totalNoAlternatives,* represent the total number of alternatives that can be generated from the attribute *alternativeSpace.*
- *tupleSpace,* defines the set of tuples that consists of elements of the dimensions.
- *alternatives,* represents the total alternatives that are derived from the *alternativeSubSpaces.*

The class *DesignSpace* consists of the following basic operations:

- *generateAlternatives()*, represents the algorithm for generating alternatives from the design space as it is stored in the attribute *alternativeSpace*. This algorithm has been explained in Figure 5.10.
- *selectAlternatives(cond: Condition)*, reduces the design space by selecting the set of design alternatives that meet the condition *cond*. The result is stored in the attribute *alternativeSpace*.
- *eliminateAlternatives(cond: Condition)*, reduces the design space by eliminating the set of design alternatives that meet the condition *cond*. The result is stored in the attribute *alternativeSpace*.
- *quantifyAlternatives(cond:Condition)*, assigns an integer value to the alternatives in the design space, based on the condition *cond*.
- *generateTupleSpace()*, generates the tuple space that consists of a set of tuples whereby each tuple consist of elements from the dimensions. The tuple space may be derived using the set operations of Cartesian product, union, difference, intersection and join. The application of the Cartesian product operation, for example, has been illustrated in section 5.3.3 on matrix-based selection, the join operation has been illustrated in section 5.4.3.
- *getTotalNoAlternatives()*, computes the number of alternatives that can be derived from the design space and stores this in the attribute *totalNoAlternatives* and returns this value.

Using this meta-model, method engineers may define different design spaces such as $S_{AdaptScheduler}$ and $S_{ObjectScheduler}$.

5.6.3 Tools Overview

In the design of the *Rumi* environment we have classified tools into *method engineering tools* and *software engineering tools*. The method engineering tools can be accessed by the *Launcher* tool that is represented in Figure 5.19.

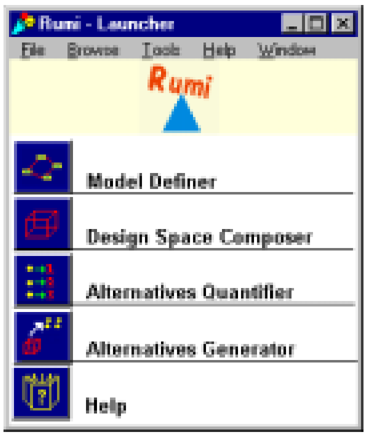
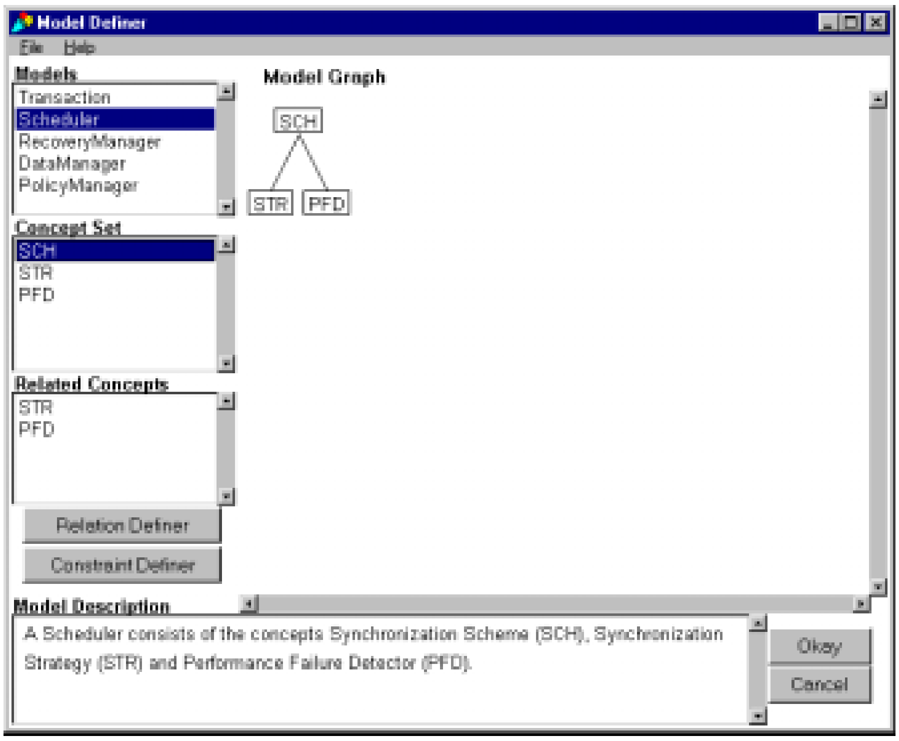


Figure 5.19 *Launcher tool of the Rumi environment*

The basic tools are *Model Definer, Design Space Composer, Alternatives Quantifier, Process Definer* and *Alternatives Generator*. These tools access a common repository in which models, property sets and design spaces are stored. We will explain these tools in the following sections.

5.6.4 Model Definer Tool

A snapshot of the tool *Model Definer* is represented in Figure 5.20. This tool is used to introduce new models such as $M_{Scheduler}$ as described in section 5.3.1. In Figure 5.20, the top-left widget *Models* is used to enter a new model into the repository. The concepts of the selected model are entered in the left-middle widget *Concept Set*. The bottom-left widget *Related Concepts* is used to define the relations between the sub-concepts of the model. The bottom widget *Model Description* is used to give general comments about the model. The right widget *Model Graph* shows a graph representation of the model, where the concepts and relations correspond to the nodes and relations, respectively. For example, the tool *Model Definer* in Figure 5.20 shows that the model $M_{Scheduler}$ is stored in the repository. Since $M_{Scheduler}$ is selected in the top-left widget, its concepts can be entered in the left-middle widget. The relations of $M_{Scheduler}$ have also been defined and the sub-concept *Sch* is related to the sub-concepts *Str* and *PFD*. The software engineer can store the model in the method base by pressing the *Okay* button.

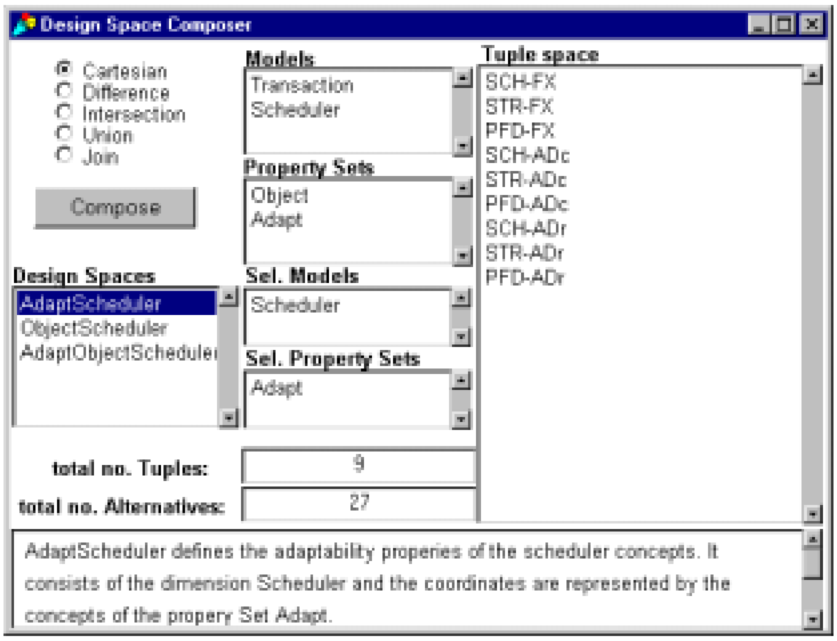


A Scheduler consists of the concepts Synchronization Scheme (SCH), Synchronization Strategy (STR) and Performance Failure Detector (PFD).

Figure 5.20 *Model Definer Tool*

5.6.5 Design Space Composer Tool

Figure 5.21 represents a snapshot of the tool *Design Space Composer*, which supports the design algebra techniques for composing design spaces as described in section 5.3.2. In Figure 5.21, the widget *Design Spaces* on the left provides a list of the design spaces in the repository. The names of the design spaces can be added, removed or updated through the widget's menu. The listbox widgets *Models* and *Property Sets* list the models and the available properties in the repository. The method engineer can select a model and a property set from these listboxes, which then appear in the listbox widgets *Sel. Models* and Sel. *Property*, respectively. In the example shown by Figure 5.21, the design space *AdaptScheduler* is selected. The tuples of the design space can be produced by pressing the button *Compose* that will then be displayed in the listbox *Tuple Space*. The radio boxes on the left enable to choose one of the operations of Cartesian product, union, difference, intersection or join for constructing the tuple space. For the design space *AdaptScheduler* the Cartesian operation has been selected.



total no. Tuples: 9
total no. Alternatives: 27

AdaptScheduler defines the adaptability properies of the scheduler concepts. It consists of the dimension Scheduler and the coordinates are represented by the concepts of the propery Set Adapt.

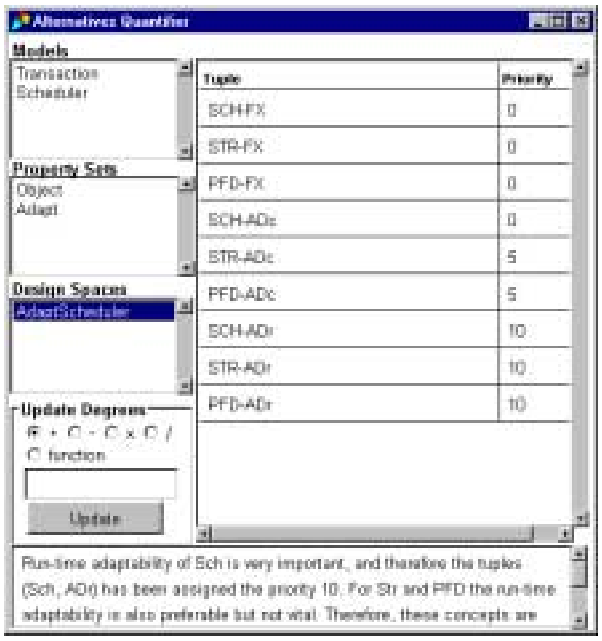
Figure 5.21 *Design Space Composer Tool*

5.6.6 Alternatives Quantifier Tool

Figure 5.22 shows the snapshot of the tool *Alternatives Quantifier* that can be used for quantifying design alternatives as it has been described in section 5.3.5. Dependent on the concept type, different priority numbers can be assigned to individual tuples of a design space by directly entering these in the fields of the column *Priority*. In the figure, the priorities for the tuples of the design space *AdaptScheduler* are shown. As described in the example of section 5.3.5, all the concepts with fixed property have been assigned the value 0 and all the concepts with run-time adaptability property have been assigned the value 10. The compile-time adaptability property of the concepts STR and PFD have been assigned the value 5 and the compile-time adaptability property of the concept SCH has been assigned the value 0.

The widget *Update Degree* provides operations for automatically computing the priorities of the tuples of the selected design space. For this, it is required that the priorities of the basic property sets such as *Object* and *Adapt* need to be defined in advance. For example, assume that the following priority values are assigned to the properties of *Adapt*: FX=0, ADc=5, ADr. Further, assume that the properties of *Object* are all assigned the value 1, indicating that these should not be considered in the prioritization of the alternatives. The software engineering can now select one of the arithmetic operations +, -, x, or / and press the *Compute* button for computing the priorities of the tuples of *AdaptScheduler*. The radio button *function* can be selected to define complex priority calculations for which an expression can be entered in the input field below. If the arithmetic operation 'x' has been selected then the tuples of

AdaptScheduler will have the values as shown in Figure 5.22, except the tuple (STR,ADc) for which the priority 5 will be computed. The automatic computation of priorities may be very useful for design spaces that include a large set of tuples.



Transaction
Scheduler

Object
Adapt

AdaptScheduler

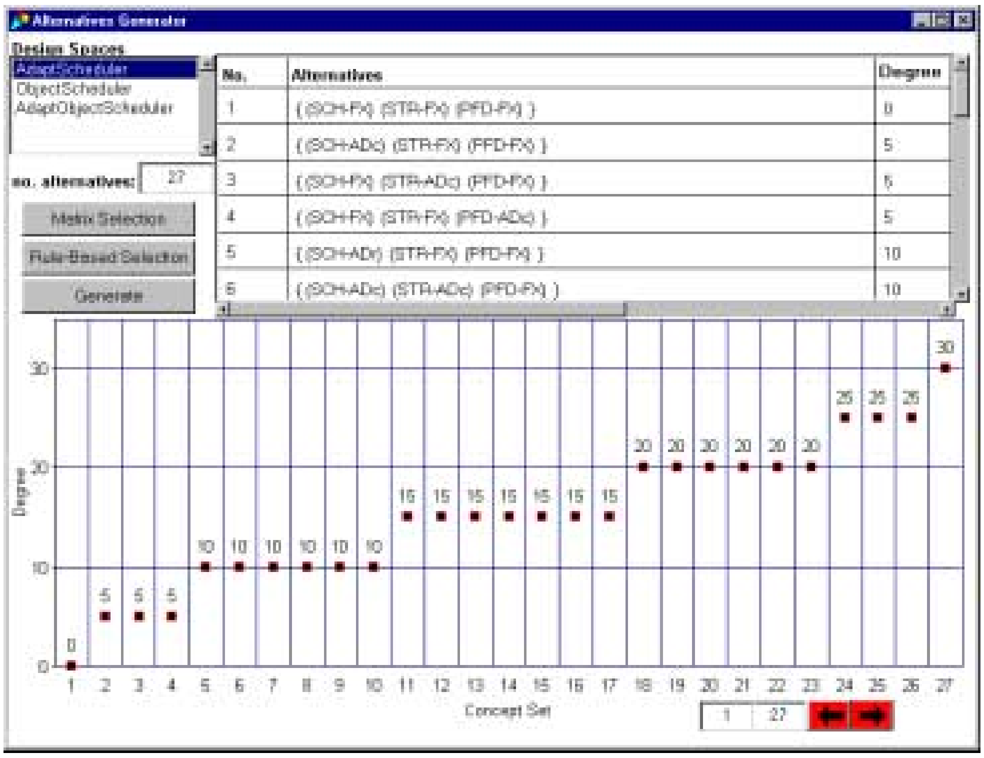
Tuple	Priority
SCH-FX	0
STR-FX	0
PFD-FX	0
SCH-ADc	0
STR-ADc	5
PFD-ADc	5
SCH-ADr	10
STR-ADr	10
PFD-ADr	10

Run-time adaptability of Sch is very important, and therefore the tuples (Sch, ADr) has been assigned the priority 10. For Str and PFD the run-time adaptability is also preferable but not vital. Therefore, these concepts are

Figure 5.22 *Alternatives Quantifier Tool*

5.6.7 Alternatives Generator Tool

To generate alternatives from the predefined design spaces the tool *Alternative Generator* is used from which a snapshot is shown in Figure 5.23. Initially, the set of alternatives for the design spaces listed in the list box *Design Spaces* is not generated. The widget *no. alternatives* defines the number of alternatives that can be derived from the selected design space. The software engineer can generate the set of alternatives by pressing the *Generate* button. Since this number of alternatives can be quite large, the tool gives an error message when the number of alternatives exceeds a predefined maximum value. If the number of alternatives is smaller than the maximum default value the alternatives will be generated and listed and ordered according to their priority values. This ordering of the alternatives is also shown in the graphic below the list of alternatives. In the graphic each point represents an alternative. The graphic shows only 30 alternatives at once. To browse the other alternatives the left and right arrows at the right corner of the window can be used.



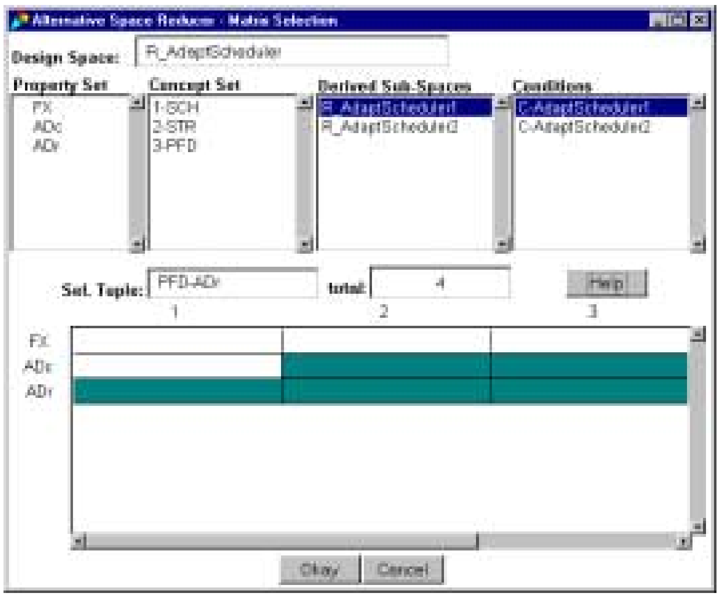
No.	Alternatives	Degree
1	{ (SCH-FX) (STR-FX) (PFD-FX) }	0
2	{ (SCH-ADc) (STR-FX) (PFD-FX) }	5
3	{ (SCH-FX) (STR-ADc) (PFD-FX) }	5
4	{ (SCH-FX) (STR-FX) (PFD-ADc) }	5
5	{ (SCH-ADr) (STR-FX) (PFD-FX) }	10
6	{ (SCH-ADc) (STR-ADc) (PFD-FX) }	10

Figure 5.23 *Alternative Generator Tool*

The software engineer can directly select some of these alternatives through the menu of the alternatives list and store this in the repository. The design space can also be reduced by either pressing the button *Matrix Selection* or the button *Rule-Based Selection* that represent selection of the alternatives through matrix-based selection and heuristic rule supported selection, respectively. We will describe the related tools in the following two sections.

Matrix-Based Alternatives Selection Tool

Figure 5.24 shows a snapshot of the *Matrix-Based Alternatives Selection* tool. The techniques for this tool have been described in section 5.3.3. In the figure an example is shown in which the design space *AdaptScheduler* is reduced. The reduced design space is called *R_AdaptScheduler* that consists of two sub-spaces *R_AdaptScheduler1* and *R_AdaptScheduler2*. The sub-space *R_AdaptScheduler1* is defined by the two conditions *C_AdaptScheduler1* and *C_AdaptScheduler2*. Conditions can be defined by selecting the cells of the matrix that is shown at the bottom of the window. The gray cells indicate the tuples that have been selected. In Figure 5.24, the condition *C_AdaptScheduler1* has been selected, which defines that the concept SCH should be run-time adaptable and the concept STR and PFD should be either compile-time or run-time adaptable. This condition results in 4 alternatives as it is shown in the figure. If the Okay button is pressed then this reduced space will be stored in the repository and with the tool *Alternatives Generator* the corresponding alternatives can be generated.



Design Space: R_AdaptScheduler

Property Set	Concept Set	Derived Sub-Spaces	Conditions
FX	1-SCH	R_AdaptScheduler1	C-AdaptScheduler1
ADc	2-STR	R_AdaptScheduler2	C-AdaptScheduler2
ADr	3-PFD		

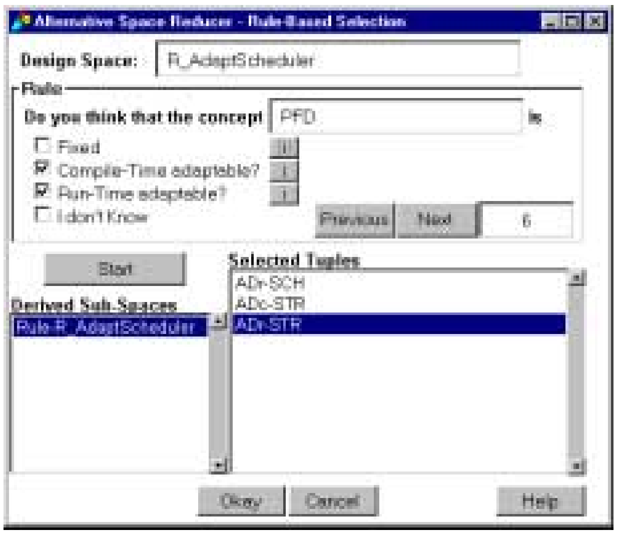
Sel. Tuple: PFD-ADr total: 4

	1	2	3
FX			
ADc		🟩	🟩
ADr	🟩	🟩	🟩

Figure 5.24 Matrix-Based Alternatives Selection Tool

Rule-Based Alternatives Selection Tool

Figure 5.25 shows a snapshot of *Rule-Based Alternatives Selection* tool. *Rumi* provides different tools for every property set. Pressing the button *Rule-Based Selection* in the *Alternatives Generation* tool in Figure 5.23, will open a tool that corresponds to the property set of the selected design space. For the design space *AdaptScheduler* a tool will be selected from the environment that is related to the property set *Adapt*. A snapshot of this tool is shown in Figure 5.25.



Design Space: R_AdaptScheduler

Rule

Do you think that the concept PFD is

- ☐ Fixed
- ☑ Compile-Time adaptable?
- ☑ Run-Time adaptable?
- ☐ I don't Know

6

Derived Sub-Spaces: Rule-R_AdaptScheduler

Selected Tuples:
ADr-SCH
ADc-STR
ADr-STR

Figure 5.25 Rule-Based Alternatives Selection Tool

This tool supports the software engineering with heuristic rules on adaptability properties of a concept set. For this, first a subspace is added in the listbox *Derived Sub-Spaces* through its menu. Pressing the *Start* button will then start the reduction of the design space through the heuristic rules support. Every rule-based selection tool typically presents a question and a set of answers that can be selected. Pressing the buttons labeled *i* will provide additional heuristic information on the presented question and answers. For example, pressing the button right to the check box *Compile-Time Adaptable* will open the following dialog box.

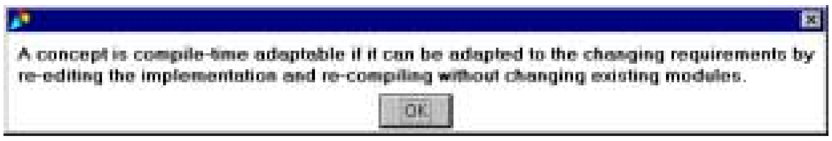

A concept is compile-time adaptable if it can be adapted to the changing requirements by re-editing the implementation and re-compiling without changing existing modules.

Figure 5.26 *Typical information dialog box for explaining heuristic rules in the Rule-Based Alternatives Selection Tool*

In the figure the reduced design space *R_AdaptScheduler* consists of a subspace *Rule-R_AdaptScheduler* for which the properties for the concepts SCH and STR have been selected. As it can be seen in the listbox *Selected Tuples* the concept SCH has been selected as run-time adaptable and the concept STR has been selected as either compile-time or run-time adaptable. At this point the number of alternatives that is still possible is 6, because PFD can still be selected in 3 ways. The current question relates to the concept PFD and the software engineer has selected it as both compile-time and run-time adaptable. The buttons *Previous* and *Next* allows the software engineer to go toe the previou step or follow with the subsequent concept.

5.7 Related Work

The notion of *design space* as we described in section 5.3.1 has been defined with the same name or similar names in several publications.

In [Lane 96] a design space is constructed from dimensions that reflect requirements and dimensions that reflect structure. Thereby, the basic objective is to identify correlations between the different dimensions of the design space so that heuristic rules may be derived and formalized that describe the appropriate and inappropriate combinations of design choices. Lane illustrates his ideas by constructing a design space for a user-interface architecture from which he extracts a set of heuristic rules that can be used to guide the software engineer in building architectures for user-interfaces.

In the domain of computer-aided design, *the morphological chart method* is described for depicting the set of possible product alternatives based on the required features

[Cross 89]. The morphological chart consists of a features dimension and a sub-solution dimension. The feature dimension includes the functions that are required for the product. The sub-solution dimension includes the means for achieving the functions. The morphological chart represents the total solution space for the product, made up of the combinations of sub-solutions. The evaluation of the alternatives is done by the *weighted objectives method*. Thereby numerical values are assigned to predefined design objectives and numerical scores for alternatives are determined with respect to these weighted objectives [Cross 89].

In [Ossher & Tarr 99] the notion of *hyperspace* is introduced for modeling software artifacts along multiple dimensions of concerns. Hyperspaces consist of multiple dimensions that group multiple disjoint concerns. A hyperspace also contains a set of *hypermodules*, which specify a set of *hyperslices* that are collections of units specified in terms of the concerns in the hyperspace.

Adaptability is generally considered as an important and desired characteristic of software systems and a number of research groups have been active in this area. For example, to improve the adaptability characteristics of software systems, the Demeter method [Lieberherr 96], Composition-Filters [Aksit 96], Aspect-Oriented Programming [Kiczales et al. 97], and Reuse Contracts [Steyaert et al. 96] are proposed as extensions to the object-oriented model. We consider these contributions important and complementary to our work. Our emphasis, however, is different. We do not propose an extension to the object-oriented model, but introduce a technique to compare the design alternatives from adaptability and performance viewpoints. In [Kiczales et al. 97], aspects are defined as properties that affect the performance or semantics of the components in systemic ways. Systemic characteristic of aspects implies that they can be considered as concepts. From this point of view, aspects are concepts, which affect the quality and/or semantics of software components. An aspect language is a language whose abstractions can directly represent one or more aspects. Aspect weaving is used to compose two or more aspects with software components. The design algebra presented in this paper can be seen as part of the aspect-oriented design. Aspects are the concepts and the weaving process is defined by the selection and elimination functions which defines different model spaces, like adaptability space and object space.

In [Jacobson et al. 97], the concept of variation point is introduced to specify locations at which variation will occur. The variation points are generally expressed using variants, which are type-like constructs. Although our adaptability modeling approach is intuitively similar, we propose an adaptability model, which can be applied along the software development process for comparing the design alternatives.

Several publications have been made on object-oriented software metrics [Chidamber & Kemerer 94]. Software metrics is quantitative measurements about any aspect of a software project. This may include *project, process and product metrics.* Product metrics aim to determine the properties of the software product, such as the amount of coupling, cohesion, code complexity, etc. Most product metrics as published in the literature are generally determined after the software system is built and there is no clear relation between the quality demands of requirements, compromises being made, and the quality of systems being built.

During the last decade, the so-called Software Performance Engineering (SPE) discipline has emerged for combining the performance analysis techniques with software engineering methods [Smith 90]. This discipline aim to construct performance models of software systems by using data about envisioned software processing. These models are used to compare software and hardware alternatives for solving performance problems. The techniques used within the context of SPE research are relevant to our work, and can be applied together with the techniques presented in this paper. Our emphasis is to compare the design alternatives both from performance and adaptability viewpoints, whereas the SPE research mainly emphasized the performance factors of the design alternatives.

Object-oriented frameworks offer well-defined infrastructures for a family of applications [Johnson & Foote 88]. Although a considerable number of successful frameworks have been developed during the last several years, it is generally agreed that designing a high-quality framework is still a difficult task [Fayad et al. 99][Taligent 96][Roberts 96]. One of the basic issues in developing object-oriented frameworks is finding the variable elements of the system which are called hot-spots [Pree 94]. The adaptability space, which we defined in this paper, can be useful for finding these hot-spots. All the elements with a tag AD in the adaptability space can be considered as being a hot-spot in the final system.

Various researchers carry out research on component oriented software development to address open systems requirements [Nierstrasz & Tsichritzis 95]. A component is defined as a static software abstraction that can be composed with other components to make an application. We think that our approach can be used to reason about and control the composition of components during software design.

5.8 Evaluation and Conclusions

There are, in general, many correct implementations of a software architecture, and each implementation may differ from the other with respect to its quality factors. Software is rarely designed for ultimate quality, but it is a compromise of multiple considerations. For example, generally the adaptability and performance factors of a software system have to be balanced. To achieve these objectives, in this chapter the following four requirements were considered important: First, to be able to compare the design alternatives, the space of the alternatives must be determined. Secondly, the alternatives must be ordered with respect to their quality factors. Thirdly, the software engineers must be able to select among the alternatives based on the requirements. Finally, the quality factors must be balanced with respect to each other.

In section 5.3 we have introduced the concept of *Design Algebra* that provides techniques to explicitly depict the design spaces of architecture specifications and defines various operations for reducing the design space and quantifying the design alternatives. A design space has been defined as a multi-dimensional space that includes the various design alternatives. The dimensions of the design space are defined by well-defined and stable concepts that have been derived from a solution domain analysis process. The reduction of the design space is applied through either the selection or the elimination of sub-spaces, optionally supported by the corresponding heuristic rules. To explicitly reason about the adaptability factors of the design alternatives formulas were introduced to depict the space of design alternatives from the adaptability point of view. Similarly, at the object-oriented modeling level formulas were defined to depict the design alternatives.

In section 5.4 we described a process for deriving adaptable object-oriented scheduler design alternatives from the architecture specification.

In section 5.5, we evaluated the time-performance quality factor for selected alternatives. To this aim, a pilot project has been carried out in which a simulation framework and a dynamic switching mechanism have been developed. The practical simulations showed that the performance computations of the alternative scheduler implementations differed and that run-time adaptable scheduler implementation alternative was effective optimizing the time performance value.

In section 5.6 we have described *Rumi*, an object-oriented tool environment that provides a set of tools for the design algebra techniques.

The techniques presented in this chapter are general and applicable to determine and balance different quality factors of design alternatives. We consider the presented techniques as initial attempts to solve the problems related to balance multiple quality factors and expect that these techniques can serve as a basis for further research in turning the software development process into an engineering activity.

Chapter 6

Conclusions

M.C.Escher's Ascending and Descending © 2000 Cordon Art-Baarn-Holland. All rights reserved.

M.C. Escher - Ascending and Descending

The illustration on the previous page shows an impossible building with a never-ending staircase. The inhabitants of these living-quarters would appear to be monks, adherents of some unknown sect. Perhaps it is their ritual duty to climb those stairs for a few hours each day. It would seem that when they get tired they are allowed to turn about and go downstairs instead of up. Yet both directions, though not without meaning, are totally useless. Although the staircase is conceptually impossible, it does not interfere with your perception of it. In fact, the paradox may not be even apparent to many people.

Software Architecture Design Analogy

Software architectures may be derived from different sources. Very often the architectural abstractions are derived from the client's requirement specifications. The derived abstractions may not interfere with the perception of the clients, though, they may be less useful for defining the stable abstractions and the architecture boundaries.

6.1 Introduction

"Focus on where you want to go, instead of where you have been."
- John M. Templeton, Worldwide Laws of Life

For each of the previous chapters, we have described the related conclusions. In this chapter we will describe the overall conclusions of the thesis that correspond to the global objectives. We describe these conclusions under the categories of conceptual foundations for mature engineering, the application of synthesis to software architecture design and the formalization of the alternative space analysis phase, which will be presented in the sections 6.2, 6.3 and 6.4, respectively. In section 6.5, we will elaborate on the future work that can extend the subjects of this thesis.

6.2 Conceptual Foundations of Mature Engineering

The initial claim of this thesis is that to solve the chronic problems of the software crisis it is necessary to view software engineering from a broad perspective. To this aim, this thesis has provided a thorough and broad analysis of software engineering from a problem-solving perspective.

To explicitly reason about the various problem-solving concepts, in chapter 2 we have presented the controlled problem-solving in context model (CPC-model), that uniquely integrates the concepts of problem-solving, control and context. The CPC-model can be utilized to analyze and evaluate different problem-solving activities. In this thesis we have utilized this model to analyze and evaluate philosophy, mature engineering and software engineering. We have described a conceptual and comparative analysis of these disciplines and provided several conclusions.

It appeared that philosophy and mature engineering both conform to the CPC model and this maturation process has been justified by a conceptual analysis from a historical perspective. In addition we have presented the mature state of these disciplines as they are applied today. It seems that the maturity of the problem solving concepts can be validated against the maturity of the control concepts in the CPC model for the corresponding discipline because the control part is directly related to an explicit understanding and interpretation of the problem solving process. In philosophy, we have seen that hermeneutic philosophy focuses on understanding of interpretation, that is control of problem solving. Hermeneutic philosophy maintains that for a correct understanding it is necessary to grasp the context of the artifacts and this observation confirms our assumption that a controlled problem solving process is valid within a given context. In traditional engineering the maturation of the problem solving process is represented by the emergence of the computer-aided design and computer-aided manufacturing techniques.

From our conceptual analysis of a historical and a project perspective of software engineering, we have concluded that software engineering is still in a pre-mature state. This is justified by the fact that it lacks several concepts that are necessary for effective problem-solving. We can basically distinguish the following important three concerns that are included in mature problem solving but to a large extent are missing in software engineering.

First, the *need* concept in the CPC model plays a basic role and as such has directed the activities of philosophy and engineering. This is to say that artificial solutions that do not directly relate to the existing needs cannot be enforced and eventually will fail to be effective. In mature engineering an explicit *technical problem analysis* phase is defined whereby the basic needs are mapped to the technical problems and organized. The technical problems are defined through an iterative process whereby both the client's perspective and the solution domain perspective is considered. The client's perspective serves to define the solution domain boundaries and the solution domain on its turn helps to identify the valid technical problems. In mature engineering, the technical problems are usually quantified to derive, for example, the overall cost of the artifact before it is actually produced.

Second, mature problem solving also includes a rich base of extensive scientific knowledge that is utilized by a *solution domain analysis phase* to derive the fundamental solution abstractions. For this we have seen that preserving and communication of this knowledge is essential, which has been shown by, for example, the different comprehensive handbooks and manuals of the mature engineering disciplines. In philosophy codification and preserving of scientific knowledge has been indicated as the basic cause for the rise of the different movements such as the Renaissance in the West.

Third, in mature problem solving different alternatives are explicitly searched from the solution domain and often organized with respect to pre-determined quality criteria. In mature engineering, for example, the quality concept plays an explicit role and the alternatives are selected in an explicit *alternative space analysis* process whereby mathematical optimization techniques such as calculus, linear programming and dynamic programming are adopted.

In mature engineering the three processes of *technical problem analysis*, *solution domain analysis* and *alternative space analysis* are integrated within the *synthesis* process. In the synthesis process the explicit problem analysis phase is followed by the search for alternatives in a solution domain that are selected based on explicit quality criteria.

In software engineering, the synthesis process is not known and the three processes are not fully integrated. We can derive the following conclusions with respect to this issue.

First, in software engineering the basic needs are derived through a requirements analysis phase that views the software system basically from the client's perspective. The client may fail to identify all the relevant problems and as such the solution may not optimally correspond to the original technical problems. Very often a distinction is made between the functional requirements and the non-functional requirements. In contrast to the mature engineering disciplines, however, there are no explicit means to quantify the requirements and constraints during the requirements analysis phase. As such the essential technical problems may not be appropriately prioritized and selected.

Second, it turns out that in software engineering solution abstractions are not derived from the solution domains. Rather, the common implicit assumption of current software engineering practices is that solution abstractions should be basically derived from the requirement specification. The general idea is that the requirement specification is specified in some form and this should be refined along the software development process until the final software is delivered. Although this view may have been sufficient and appropriate for the early well-defined numerical calculation problems in the 1940s, it does not suffice to derive the fundamental solution abstractions for the currently required large and complex software systems.

One may argue that the reason for the lack of integration of a solution domain analysis in software engineering is mainly due to the fact that the corresponding solution domain knowledge is not as rich as that in mature engineering where the basic theories have been matured over several centuries. This should not be regarded, however, as the fundamental cause for not applying a solution domain analysis process in the software development process. As a matter of fact computer science, a basic solution domain knowledge of software engineering, has contributed to the codification and organization of some relevant theories but current software engineering practices proceed almost independently of this organized knowledge. Moreover, it should be noted that solution domain analysis has also already been introduced in the last decade, though, these are not sufficiently integrated in the current software development methods and remain as a separate field.

Our third observation that we derived from our conceptual and comparative analysis is related to the management of alternatives in software engineering. In contrast to mature engineering, software engineering does not seem to have an explicit concept of alternative or alternative space analysis process. The alternatives for a given design problem are usually selected based on experience, trial-and-error and intuition. Moreover, quantification of the design alternatives for evaluation is definitely not common in software engineering.

6.3 Application of Synthesis to Software Architecture Design

Obviously, the concept of synthesis plays a fundamental role in mature problem solving and for improving the maturity of software engineering it is necessary to integrate this concept synthesis within the current software engineering practices.

This thesis focused on the software architecture design phase of the software engineering process and attempted to integrate the synthesis concept herein. The software architecture design phase represents the most crucial part of the software development process because it defines the overall structure of the software system and the subsequent phases are strongly dependent on the result of this phase. It is thus of utmost importance to define the right software architecture that provides solutions with the appropriate quality factors for the relevant technical problems. Applying the synthesis process to software architecture design may provide substantial support for this.

In chapter 3, we have provided a conceptual analysis on the state-of-the-art architecture design approaches and classified these according to their source of their basic solution abstractions. As such we distinguished between artifact-driven, use-case driven, domain-driven and pattern-driven software architecture design approaches. We have provided explicit models for these classes of approaches, evaluated these approaches and identified their basic problems in providing stable architectures. The fundamental problems that we derived were the difficulties in planning the architectural design phase, difficulties in finding the stable architectural abstractions, difficulties in leveraging the problem, the poor semantics of the identified architectural abstractions and finally the weak support for composing architectural abstractions.

It turned out that an important concern in software architecture design is the identification of the fundamental architectural abstractions. This relates to understanding of the relevant technical problems and deriving the stable solution abstractions. Thereby the software engineer may need to consider different solution alternatives. Software architecture design thus inherently needs to provide the concepts of the synthesis process.

In chapter 4, we have applied the synthesis concept to the software architecture design process for solving the identified problems of the architecture design approaches. This resulted in a novel approach that we termed *synthesis-based software* *architecture design approach*[43]. This approach includes the explicit processes of technical problem analysis, solution domain analysis and alternative space analysis that are important for finding the stable architectural abstractions. During the technical problems analysis the initial requirement specifications are mapped to relevant technical problems. In the solution domain analysis, for each technical problem the necessary solution domains are identified and solution domain concepts are extracted by identifying commonalties and variabilities of the extracted knowledge from the solution domain. The solution domain concepts are mapped to the components of the conceptual architecture. In the alternative space analysis process, for each solution domain concept the set of possible alternative instances are depicted and the constraints among these are defined. This determined the adaptability of the architecture.

We have illustrated the approach for the design of an atomic transaction system architecture for a distributed car dealer information system. Hereby the software architecture had to represent various transaction protocols, such as concurrency control and recovery, that could be easily customized to the various needs of the different dealers in different countries. In addition, to provide optimal performance, the architecture had to include mechanisms for run-time adaptability of the transaction protocols. During the technical problem analysis phase we could derive and organize the relevant technical problems for the project. From the solution domain analysis on transaction theory the concepts for the atomic transaction architecture were derived. In the alternative space analysis process we could derive the constraints among the transaction concepts and as such determined the adaptability of the architecture. We validated the transaction architecture using the theories on atomic transactions.

6.4 Formalizing Alternative Space Analysis Phase: Design Algebra

The architecture synthesis process as described in chapter 4 provides an explicit process for reasoning about the various alternatives that an architectural component should represent. Chapter 5 focused on the realization of the architecture using object-oriented analysis and design methods that help software engineers to express their solutions in terms of object-oriented abstractions. Architectures can be realized

[43] Various experimental synthesis techniques have been applied in the TRESE group in different projects over the last seven years and these experiences provided a useful basis for the synthesis-based software architecture design approach that has been described in this thesis.

in many different ways and each alternative may differ with respect to the various quality factors such as adaptability, performance and reusability. Current object-oriented analysis and design methods do not provide explicit means for depicting the set of alternative designs that can be derived from a given conceptual architecture. The design alternatives are usually selected based on the experience and intuition of the software engineer. In chapter 5, we have introduced a formalism, called *design algebra*, that provides techniques to explicitly depict the set of design alternatives and prioritize and select alternatives based on quality criteria. A design space has been defined as a multi-dimensional space that includes all the possible design alternatives. The dimensions of the design space are defined by well-defined and stable concepts that have been derived from a solution domain analysis process. Since design spaces can be very large to depict all the alternatives, design algebra provides techniques for reducing the design space that can be applied through either the selection or the elimination of sub-spaces, optionally supported by the corresponding heuristic rules. To explicitly reason about the adaptability factors of the design alternatives, formulas were introduced to depict the space of design alternatives from the adaptability point of view. Similarly, at the object-oriented modeling level formulas were defined to depict the design alternatives.

We have illustrated the design algebra techniques for deriving adaptable object-oriented designs of the concept scheduler in the atomic transaction architecture and gradually derived the set of alternatives based on the adaptability and performance factors. The techniques of design algebra are general and can be integrated in the current object-oriented analysis and design methods.

6.5 Further Work

This work can be elaborated in many different ways. In the following we provide some interesting and relevant directions.

The CPC model has been utilized to analyze software engineering, traditional engineering and philosophy, but it may likewise be used to analyze other problem solving activities. It may for example be utilized to analyze sociology from a problem-solving perspective. This may be important for software engineering, because software engineering directly effects the society and society on its turn impacts software engineering. It would be worthwhile to highlight the interplay between these two disciplines.

We have applied the synthesis concept at the software architecture design phase. It would be worthwhile to consider applying this concept on the other phases of software development process, because every phase basically has to deal with the identification of the right abstractions and the management of different alternatives. Synthesis at the architecture design phase can then be considered as high-level synthesis because it has to deal with high-level abstractions, synthesis at the design and implementation phases can be considered as low-level synthesis.

Design algebra provides useful techniques to depict the space of design alternatives and for selecting appropriate alternatives based on quality factors. We have illustrated the use of design algebra techniques for the atomic transaction architecture. It is interesting to apply it for other applications and analyze the results. We have considered adaptability and time-performance as the basic quality factors for selecting design alternatives. Obviously, analyzing other quality factors such as reusability may be useful and it would be worthwhile to express these in design algebra as well. Design algebra provides techniques for quantifying alternatives. For effective optimization of the various alternatives based on the quality factors it is beneficial to apply optimization techniques such as linear and dynamic programming and calculus.

We have derived several lessons and concepts from the analysis in chapter 2. In this thesis we have focused on the concept of synthesis and successfully applied this to software architecture design. The analysis, however, has also resulted in several other concepts and lessons that may be worthwhile to integrate in software engineering as well. In hermeneutic philosophy, for example, it is 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 author. The software development process is decomposed in various phases and usually the user of an artifact in one phase may be different from its producers in the previous phase. For example, a software architect will typically provide the software architecture to the software engineers in the analysis and design phases, that on their turn will provide analysis and design models to the implementers of the artifact. In all these transformation processes interpretation plays a fundamental role. Therefore, integration of the concepts of hermeneutics in software engineering may provide useful techniques for the traceability and maintainability of software systems.

References

Chapter 1 - Introduction

[Agre 82] Agre, G.P. *The concept of problem*, Educational Studies, vol. 13, pp. 121-142, 1982.

[Gibbs 94] Gibbs, W.W., *Software's Chronic Crisis*. The Scientific American, pp. 86-95, September, 1994.

[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 Clifss, NJ, 1976.

[Rubinstein & Pfeiffer 80] Rubinstein, M.F. & Pfeiffer, K. *Concepts in Problem solving*. Prentice-Hall, Englewood Cliffs, 1980.

Chapter 2 - On the Notion of Software Engineering: Problem Solving Perspective

[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.

[Cooke 97] Cooke, R. *The history of mathematics : a brief course*, New York: Wiley, 1997.

[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 67] Knuth, D.E. *The Art of Computer Programming*, Addison-Wesley, 1967.

[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.

[Kuhn 62] Kuhn, T. *The Structure of Scientific Revolutions*, University of Chicago Press, Chicago, 1962.

[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 & Braha 96] 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.

[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 Clifss, NJ, 1976.

[Norman 98] Norman, D. A. *The invisible computer*. Cambridge, MA: MIT Press, 1998.

[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 Mod*ules, Communications of the ACM, 15 (12), 1972.

[Parnas 76] Parnas, D.L. *On the design and development of program families*. In IEEE Transactions on Software Engineering, Vol. SE-2, no. 1, pp. 1-9, 1976.

[Popper 34] Popper, K.R. *Logic of Scientific Discovery*, London: Hutchinson, 1934.

[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. 91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[Rumbaugh et al. 98] Rumbaugh, J., Jacobson, I., & Booch, G. *The Unified Modeling Language Reference Manual*, Addision-Wesley, 1998.

[Rombach et al. 93] Rombach, H.D., Basili, V.R., & Selby, R.W. (Eds.) *Experimental Software Engineering Issues: Critical Assessment and Future Directions,* International Workshop Dagstuhl Castle, Germany, Lecture Notes in Computer Science, Vol. 706, Springer Verlag, 1993.

[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.

[Sommerville 95] Sommerville, I. *Software Engineering.* Addison-Wesley, 1995.

[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.

Chapter 3 - Classification and Evaluation of Software Architecture Design Approaches

[Abowd et al. 94] Abowd, G.; Bass, L.; Kazman, R.; & Webb, M. *SAAM: A Method for Analyzing the Properties of Software Architectures,* 81-90. Proceedings of the 16th International Conference on Software Engineering. Sorrento, Italy, May 16-21, 1994. Los Alamitos, CA: IEEE Computer Society Press, 1994.

[Aksit et al. 00] Aksit, M., Bergmans, L., Berg van den K., Broek, van den P., Rensink, A., Noutash, A., & Tekinerdogan, B. *Towards Quality-Oriented Software Engineering,* to be published in Software Architectures and Component Technology: The State of the Art in Research and Practice, M. Aksit (Ed.), Kluwer Academic Publishers, January 2000.

[Aksit & Bergmans 92] Aksit M. & Bergmans L. *Obstacles in Object-Oriented Software Development*, Proceedings OOPSLA '92, ACM SIGPPLAN Notices, Vol. 27, No. 10, pp. 341-358, October 1992.

[Alexander 79] Alexander C., Ishikawa S., & Silverstein M. *A Pattern Language.* New York City: Oxford University Press, 1979.

[Arrango 94] Arrango, G. *Domain Analysis Methods.* In *Software Reusability*, Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds.), Ellis Horwood, New York, New York, pp. 17-49, 1994.

[Bass et al. 97a] Bass, L., Clements, P., Cohen, S., Northrop, L. & Withey, J. *Product Line Practice Workshop Report*, Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Bass et al. 97b] Bass, L., Clements, P., Chastek, G., Cohen, S., Northrop, L.,. & Withey, J. *2nd Product Line Practice Workshop Report*, Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Bass et al. 98] Bass, L., Clements, P., & Kazman, R. *Software Architecture in Practice,* Addison-Wesley 1998.

[Bernstein & Newcomer 97] Bernstein, P.A., & Newcomer, E. *Principles of Transaction Processing,* Morgan Kaufman Publishers, 1997.

[Booch 91] Booch, G. *Object-Oriented Design with Applications,* The Benjamin/Cummings Publishing Company, Inc, 1991.

[Buschmann et al. 96] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns,* John Wiley & Sons, 1996.

[Clements 96] Clements, P. *A Survey of Architectural Description Languages,* Proceedings of the 8th International Workshop on Software Specification and Design, Paderborn, Germany, March, 1996.

[Clements & Northrop 96] Clements, P.C., & Northrop, L.M., *Software Architecture: An Executive Overview*, Technical Report, CMU/SEI-96-TR-003, Carnegie Mellon University, 1996.

[Clements et al. 85] Clements, P.; Parnas, D.; & Weiss, D. *The Modular Structure of Complex Systems*. IEEE Transactions on Software Engineering SE-11, 1, pp. 259-266, 1985.

[Coplien 92] Coplien, J.O. *Advanced C++ -Programming Styles and Idioms*, Addison-Wesley, Reading, MA, 1992.

[Czarnecki 99] Czarnecki, C., *Generative Programming*: *Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*, PhD Thesis, Technical University of Ilmenau, 1999.

[Date 90] Date, C.J. *An Introduction to Database Systems*, Vol. 3, Addison Wesley, 1990.

[Dijkstra 68] Dijkstra, E.W. *The Structure of the'T.H.E.' Mulitprogramming System*. Communications of the ACM 18, 8 , pp. 453-457, 1968.

[Elmagarmid 91] Elmagarmid, A.K. (ed.) *Transaction Management in Database Systems*, Morgan Kaufmann Publishers, 1991.

[Fowler 96] Fowler, M. *Analysis Patterns : Reusable Object Models*, Addison-Wesley, 1996.

[Gamma et al. 95] Gamma, E.; Helm, R.; Johnson, R.; & Vlissides, J. *Design Patterns, Elements of Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[Garlan & Shaw 93] Garlan, D. & Shaw, M. *An Introduction to Software Architecture*. Advances in: Software Engineering and Knowledge Engineering. Vol 1. River Edge, NJ: World Scientific Publishing Company, 1993.

[Garlan et al. 95] Garlan, D., Allen, R., & Ockerbloom, J. *Architectural Mismatch: Why It's Hard to Build Systems Out of Existing Parts*, 170-185. Proceedings, 17th International Conference on Software Engineering. Seattle, WA, April 23-30, 1995. New York: Association for Computing Machinery, 1995.

[Gomaa 92] Gomaa, H. *An Object-Oriented Domain Analysis and Modeling Method for Software Reuse*. In *Proceedings of the Hawaii International Conference on System Sciences*, Hawaii, January, 1992.

[Hayes-Roth 94] Hayes-Roth, F. *Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program*, 1994.

[Howard 87] Howard, R.W. *Concepts and Schemata*: *An Introduction*, Cassel Education, 1987.

[Jacobson et al. 99] Jacobson, I., Booch, G., & Rumbaugh, J., *The Unified Software Development Process*, Addison-Wesley, 1999.

[Kang et al. 90] Kang, K., Cohen, S., Hess, J., & Nowak, W., & Peterson, S. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990

[Kruchten 95] Kruchten, Philippe B. *The 4+1 View Model of Architecture.* IEEE Software, Vol 12, No 6, pp. 42-50, November 1995.

[Lakoff 87] Lakoff, G. *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind,* The University of Chicago Press, 1987.

[Parnas 72] Parnas, D. *On the Criteria for Decomposing Systems into Modules.* Communications of the ACM 15, 12 (December 1972): 1053-1058.

[Parnas 76] Parnas, D. *On the Design and Development of Program Families.* IEEE Transactions on Software Engineering SE-2, 1: 1-9, 1976.

[Parsons & Wand 97] Parsons, J., & Wand, Y. *Choosing Classes in Conceptual Modeling,* Communications of the ACM, Vol 40. No. 6., pp. 63-69, 1997

[Perry & Wolf 92] Perry, D.E. & Wolf, A.L. *Foundations for the Study of Software Architecture.* Software Engineering Notes, ACM SIGSOFT 17, 4: 40-52, October 1992.

[PLOP 97] Proceeding of the *Pattern Languages of Programs Conference.*

[Pree 95] Pree, W. *Design Patterns for Object-Oriented Software Development,* Addison-Wesley, 1995.

[Prieto-Diaz & Arrango 91] Prieto-Diaz, R., & Arrango, G. (Eds.). *Domain Analysis and Software Systems Modeling.* IEEE Computer Society Press, Los Alamitos, California, 1991.

[Rosch et al. 76] Rosch , E., Mervis, C.B., Gray, W.D., Johnson, D.M., and Boyes-Braem, P. *Basic objects in natural categories.* Cognitive Psychology 8: 382-439, 1976.

[Rumbaugh et al. 91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. *Object-Oriented Modeling and Design,* Prentice Hall, 1991.

[SEI 00] Carnegie Mellon Software Engineering Institute, Web-site: http://www.sei.cmu.edu/architecture/, 2000.

[Shaw 95] Shaw, M. *Making Choices: A Comparison of Styles for Software Architecture.* IEEE Software 12, 6 27-41, November, 1995.

[Shaw 98] Shaw, M. *Moving from Qualities to Architectures: Architectural Styles,* in: L. Bass, P. Clements, & R. Kazman (eds.), Software Architecture in Practice, Addison-Wesley, 1998.

[Shaw & Clements 97] Shaw, M., & Clements, P. *A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems,* Proc. COMPSAC97, 1st Int'l Computer Software and Applications Conference, August, 1997.

[Shaw & Garlan 96] Shaw, M. & Garlan, D. *Software Architectures: Perspectives on an Emerging Discipline,.* Englewood Cliffs, NJ: Prentice-Hall, 1996.

[Simos et al. 96] Simos, M., Creps, D., Klinger, C., Levine, L., & Allemang, D. *Organization Domain Modeling (ODM) Guidebook,* Version 2.0. Informal Technical Report for STARS, STARS-VC-A025/001/00, June 14, http://www.synquiry.com, 1996.

[Smith & Medin 81] Smith, E.E., & Medin, D.L., *Categories and Concepts*, Harvard University Press, London, 1981.

[Stillings et al. 95] Stillings, N.A., Weisler, S.E., Chase, C.H., Feinstein, M.H., Garfield, J.L., & Rissland, E.L., *Cognitive Science: An Introduction*. Second Edition, The MIT Press, Cambridge, Massachusetts, 1995.

[Soni et al. 95] Soni, D., Nord, R., Hofmeister, C. *Software Architecture in Industrial Applications*. 196-210. Proceedings of the 17th International ACM Conference on Software Engineering, Seattle, WA, 1995.

[Tracz & Coglianese 92] W. Tracz and L. Coglianese. *DSSA Engineering Process Guidelines. Technical Report*, ADAGE-IBM-9202, IBM Federal Systems Company, December, 1992.

[Tracz 95] Tracz, W. *DSSA (Domain-Specific Software Architecture) Pedagogical Example*. In *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 4, July 1995.

[Wartik & Prieto-Diaz 92] Wartik, S., & Prieto-Díaz, R. Criteria for Comparing Domain Analysis Approaches. In International Journal of Software Engineering and Knowledge Engineering, vol. 2, no. 3, pp. 403-431, September 1992.

[Webster 00] Merriam Webster on-line Dictionary, http://www.m-w.com/cgi-bin/dictionary, 2000.

[Wittgenstein 53] Wittgenstein, L. *Philosophical investigations*, Macmillan, New York, 1953.

Chapter 4 - Architecture Synthesis Process

[Agrawal 87] Agrawal, R., Carey, M., & Livney, M. *Concurrency control performance modelling: Alternatives and implications.* ACM Transactions on Database Systems, Vol. 12, No. 4, pp. 609-654, December 1987.

[Ahsmann & Bergmans 95] Ahsmann F & Bergmans L. *I-NEDIS: New European Dealer System*, Project plan I-NEDIS, 1995.

[Aksit 00] Aksit, M. *Course Notes: Design Software Architectures.* Post-Academic Organization, 2000.

[Aksit et al. 98] Aksit, M., Marcelloni, F., & Tekinerdogan B. *Developing Object-Oriented Frameworks using domain models*, ACM computing surveys, 1998.

[Aksit et al. 99] Aksit, M., Tekinerdogan, B., Marcelloni, F., & Bergmans, L. *Deriving Object-Oriented Frameworks from Domain Knowledge.* in M. Fayad, D. Schmidt, R. Johnson (eds.), Building Application Frameworks: Object-Oriented Foundations of Framework Design, Wiley & Sons, 1999.

[Aksit et al. 96] Aksit, M., Tekinerdogan, B, & Bergmans, L. *Achieving adaptability through separation and composition of concerns*, in Max Muhlhauser (ed), Special issues in Object-Oriented Programming, Workshop Reader of the 10th European Conference on Object-Oriented Programming, ECOOP '96, Linz, Austria, July, 1996.

[Arend 99] Arend, E. van der. *Design of an Architecture for a Quality Management Push Framework.* MSc thesis, Dept. of Computer Science, University of Twente, 1999.

[Arrango 94] Arrango, G. *Domain Analysis Methods.* In *Software Reusability*, Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds.), Ellis Horwood, New York, New York, pp. 17-49, 1994.

[Atkins & Coady 92] Atkins, M.S. & Coady, M.Y. *Adaptable Concurrency Control for Atomic Data Types.* ACM Transactions on Computer Systems, Vol. 10, No. 3, pp. 190-225, August 1992.

[Barghouti & Kaiser 91] Barghouti, N.S., & Kaiser, G.E. *Concurrency Control in Advanced Database Applications*, ACM Computing Surveys, Vol. 23, No. 3, September, 1991.

[Bass et al. 98] Bass, L., Clements, P., & Kazman, R. *Software Architecture in Practice*, Addison-Wesley 1998.

[Bernstein & Goodman 83] Bernstein, A., & Goodman, N. *Concurrency Control in Distributed Database Systems*, ACM Transactions on Database Systems, 8(4): 484-502, 1983.

[Bernstein & Newcomer 97] Bernstein, P.A., & Newcomer, E. *Principles of Transaction Processing*, Morgan Kaufman Publishers, 1997.

[Bernstein et al. 87] Bernstein, P.A., Hadzilacos, V., & Goodman, N. *Concurrency Control & Recovery in Database Systems,* Addison Wesley, 1987.

[Bhargava 87] Bhargava, B.K. editor. *Concurrency Control and Reliability in distributed Systems,*Van Nostrand Reinhold, 1987.

[Booch et al. 99] Booch, G., Jacobson, I., & Rumbaugh, J. *The Unified Modeling Language User Guide,* Addison-Wesley, 1999.

[Bourque et al. 99] Bourque, P., Dupuis, R., Abran, A., Moore, J.W., & Tripp, L. *The Guide to the Software Engineering Body of Knowledge,* Vol. 16, No. 6, pp. 35-45, November/December, 1999.

[Buschmann et al. 99] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns,* John Wiley & Sons, 1999.

[Carey 84] Carey, M., & Stonebraker, M. *The performance of concurrency control algorithms for database management systems.* In Proceedings of the 10th International Conference on Very Large Data Bases, Singapore, pp. 107-118, 1984.

[Cellary et al. 89] Cellary, W., Gelenbe, E., & Morzy, T. *Concurrency Control in Distributed Database Systems,* North-Holland Press, 1989.

[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 and Sons, 1989.

[Date 90] Date, C.J. *An Introduction to Database Systems,* Vol. 3, Addison Wesley, 1990.

[Diaper 89a] Diaper, D. (ed.). *Knowledge Elicitation,* Ellis Horwood, Chichester, 1989.

[Diaper 89b] Diaper, D. *Task Analysis for Human Computer Interaction,* Wiley & Sons, 1989.

[Dorf & Bishop 98] Dorf, R.C., & Bishop, R.H. *Modern Control Systems.* Addison-Wesley, 1998.

[Elmagarmid 92] Elmagarmid, A.K. editor. *Database Transaction Models for AdvancedApplications Transaction Management in Database Systems,* Morgan Kaufmann Publishers, 1992.

[Firlej & Hellens 91] Firlej, M., & Hellens, D. *Knowledge elicitation: a practical handbook,* New York, Prentice Hall, 1991.

[Foerster 79] Foerster, H. Von., *Cybernetics of Cybernetics,* in: Klaus Krippendorff (ed.), Communication and Control in Society, New York: Gordon and Breach, 1979.

[Gajski et al. 92] Gajski, D.D., Dutt, N.D., Wu, A., & Lin, S. *High-level synthesis : introduction to chip and system design,* Boston : Kluwer Academic Publishers, 1992.

[Glass & Vessey 95] Glass, R.L., & Vessey, I. *Contemporary Application-Domain Taxonomies*, IEEE Software, Vol. 12, No. 4, July 1995.

[Gonzales & Dankel 93] Gonzalez, A.J., & Dankel, D.D. *The Engineering of Knowledge-Based Systems*, Prentice Hall, Englewood Cliffs, NJ, 1993.

[Gray & Reuter 93] Gray, J., & Reuter, A. *Transaction processing: concepts and techniques*, San Mateo, Morgan Kaufmann Publishers 1993.

[Guerraoui 94] Guerraoui, R. *Atomic Object Composition*. In Proceedings of the European Conference on Object-Oriented Programming, LNCS 821, Springer-Verlag, pp. 118-138, 1994.

[Hadzilacos 88] Hadzilacos, V. *A theory of reliability in Database Systems*, Journal of the ACM, 35(1): 121-145, January 1988.

[Haerder & Reuter 83] Haerder, T., & Reuter, A. *Principles of Transaction-Oriented Database Recovery*. ACM Computing Surveys, Vol. 15. No. 4. pp. 287-317, 1983.

[Highleyman 89] Highleyman, W.H. *Performance analysis of transaction processing systems*, Englewood Cliffs, NJ : Prentice Hall, 1989.

[Howard 87] Howard, R.W. *Concepts and Schemata*: *An Introduction*, Cassel Education, 1987.

[Jacobson et al. 99] Jacobson, I., Booch, G., & Rumbaugh, J., *The Unified Software Development Process*, Addison-Wesley, 1999.

[Jajodia & Kerschberg 97] Jajodia, S., & Kerschberg, L. *Advanced Transaction Models and Architectures*, Boston: Kluwer Academic Publishers, 1997.

[Kruchten 95] Kruchten, Philippe B. *The 4+1 View Model of Architecture*. IEEE Software, Vol 12, No 6, pp. 42-50, November 1995.

[Kumar 96] Kumar, V. *Performance of Concurrency Control Mechanisms in Centralized Database Systems*. Prentice-Hall, 1996.

[Lakoff 87] Lakoff, G. *Women, Fire, and Dangerous Things*: *What Categories Reveal about the Mind*, The University of Chicago Press, 1987.

[Lieberherr 96] Lieberherr, K.J. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston, 1996.

[Loucopoulos & Karakostas 95] Loucopoulos, P., & Karakostas, V. *System requirements engineering*, London [etc.] : McGraw-Hill, 1995.

[Lynch et al. 94] Lynch, N., Merrit, M., Weihl, W., & Fekete, A. *Atomic Transactions*. Morgan Kaufmann Publishers, 1994.

[Maher 90] Maher, M.L., *Process Models for Design Synthesis*, AI-Magazine, pp. 49-58, Winter 1990.

[Maimon & Braha 96] Maimon, O., & Braha, D. *On the Complexity of the Design Synthesis Problem*, IEEE Transactions on Systems, Man, And Cybernetics-Part A: Systems and Humans, Vol. 26, No. 1, January 1996.

[Meyer & Booker 91] Meyer, M., & Booker, J. *Eliciting and Analyzing Expert Judgment: A practical Guide, Volume 5 of Knowledge-Based Systems*, London: Academic Press, 1991.

[Moss 85] Moss, J.E.B. *Nested Transactions : an approach to reliable distributed computing*, Cambridge, MA: MIT Press, 1985.

[Newell & Simon 76] Newell, A., & Simon, H.A., *Human Problem Solving*, Prentice-Hall, Englewood Clifss, NJ, 1976.

[Papadimitriou 86] C.H. Papadimitriou. *The theory of Database Concurrency Control*. Computer Science Press, 1986.

[Parsons & Wand 97] Parsons, J., & Wand, Y. *Choosing Classes in Conceptual Modeling*, Communications of the ACM, Vol 40. No. 6., pp. 63-69, 1997

[Partridge & Hussain 95] Partridge, D., & Hussain, K.M. *Knowledge-Based Information Systems*, McGraw-Hill, 1995.

[Prieto-Diaz & Arrango 91] Prieto-Diaz, R., & Arrango, G. (Eds.). *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, Los Alamitos, California, 1991.

[Polya 57] Polya, G. *How to solve it : a new aspect of mathematical method*, New York, Doubleday, 1957.

[Pu et al. 88] Pu, C., Kaiser, G., & Hutchinson, N. *Split-transactions for open-ended activities*. In Proceedings of the 14th VLDB Conference, 1988.

[Reich & Fenves 91] Reich, Y., & Fenves, S.J. *The formation and use of abstract concepts in design*, in: Concept Formation: Knowledge and Experience in Unsupervised Learning, D.H.J. Fisher, M.J. Pazzani, & P. Langley (eds.), Los Altos, CA, pp. 323--353, Morgan Kaufmann, 1991.

[Roxin 97] Roxin, E.O. *Control theory and its applications*. Amsterdam, Gordon and Breach Science Publishers, 1997.

[Rubin 98] Rubin, R. *Foundations of library and information science*. New York, Neal-Schuman, 1998.

[Shaw & Garlan 96] Shaw, M. & Garlan, D. *Software Architectures: Perspectives on an Emerging Discipline*,. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[Shaw 98] Shaw, M. *Moving from Qualities to Architectures: Architectural Styles*, in: L. Bass, P. Clements, & R. Kazman (eds.), Software Architecture in Practice, Addison-Wesley, 1998.

[Shinners 98] Shinners, S.M. *Modern Control System Theory and Design*. Wiley, 1998.

[Stillings et al. 95] Stillings, N.A., Weisler, S.E., Chase, C.H., Feinstein, M.H., Garfield, J.L., & Rissland, E.L., *Cognitive Science: An Introduction*. Second Edition, The MIT Press, Cambridge, Massachusetts, 1995.

[Sommerville & Sawyer 97] Sommerville, I., & Sawyer, P. *Requirements engineering: a good practice guide*, Chichester, Wiley, 1997.

[Stillings et al. 95] Stillings, N.A., Weisler, S.E., Chase, C.H., Feinstein, M.H., Garfield, J.L., & Rissland, E.L., *Cognitive Science: An Introduction*. Second Edition, The MIT Press, Cambridge, Massachusetts, 1995.

[Tekinerdogan 94] Tekinerdogan, B. *Design of an object-oriented framework for atomic transactions*, MSc. thesis, University of Twente, Dept of Computer Science, 1994.

[Tekinerdogan 95a] Tekinerdogan, B., *Overall Requirements Analysis for INEDIS*, Siemens-Nixdorf/University of Twente, INEDIS project, 1995.

[Tekinerdogan 95b] Tekinerdogan, B. *Requirements for Transaction Processing in INEDIS*, Siemens-Nixdorf/University of Twente, INEDIS project, 1995.

[Tekinerdogan 96] Tekinerdogan, B. *Reliability problems and issues in a distributed car dealer information system*, INEDIS project, 1996.

[Tekinerdogan & Aksit 97] Tekinerdogan, B., & Aksit, M. *Adaptability in object-oriented software development*, Workshop report, in M. Muhlhauser (ed), Special issues in Object-Oriented Programming, Dpunkt, Heidelberg, 1997.

[Tekinerdogan & Aksit 99] Tekinerdogan, B., & Aksit, M. *Deriving design aspects from conceptual models*. In: Demeyer, S., & Bosch, J. (eds.), Object-Oriented Technology, ECOOP '98 Workshop Reader, LNCS 1543, Springer-Verlag, pp. 410-414, 1999.

[Tracz & Coglianese 92] W. Tracz and L. Coglianese. *DSSA Engineering Process Guidelines. Technical Report*, ADAGE-IBM-9202, IBM Federal Systems Company, December, 1992.

[Thayer et al. 97] Thayer, R.H., Dorfman, M., & Bailin, S.C. *Software requirements engineering*, Los Alamitos, IEEE Computer Society Press, 1997.

[Traiger et al. 82] Traiger, I.L., Gray, J., Caltiere, C.A., & Lindsay, B.G. *Transactions and Consistency in Distributed Database Systems*, ACM Transactions on Database Systems, Vol. 7, No. 3, September 1982, pp 323-342.

[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.

[Vuijst 94] Vuijst, C. *Design of an Object-Oriented Framework for Image Algebra*. MSc thesis, Dept. of Computer Science, University of Twente, 1994.

[Warmer & Kleppe 99] Warmer, J.B., & Kleppe, A.G. *The Object Constraint Language : Precise Modeling With Uml*, Addison-Wesley, 1999.

[Wartik & Prieto-Diaz 92] Wartik, S., & Prieto-Díaz, R. *Criteria for Comparing Domain Analysis Approaches*. In International Journal of Software Engineering and Knowledge Engineering, vol. 2, no. 3, pp. 403-431, September 1992.

[Weihl 89] Weihl, W. *The impact of recovery on concurrency control*. Proceedings of the eigth ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems March 29 - 31, Philadelphia, PA USA, 1989.

[Weihl 90] Weihl, W.E. *Linguistic support for atomic data types.* ACM Transactions on Programming Languages and Systems, Vol. 12, No. 2, 1990.

[Wielinga et al. 92] Wielinga, B.J., Schreiber, T., & Breuker, J.A., *KADS: a modeling approach to knowledge engineering,* Academic Press, 1992.

[Willems 98] Willems, R. *Ontwikkelen van verzekeringsproducten*, dutch, translation: Development of Insurance Products, MSc thesis, Dept. of Computer Science, University of Twente, 1999.

[Wu et al. 95] Wu, Z., Stroud, R.J., Moody, K., & Bacon, J. *The design and implementation of a distributed transaction system based on atomic data types*, Distributed Syst, Engineering, 2, pp. 50-64, 1995.

Chapter 5 - Balancing Architecture Implementation Alternatives

[Agrawal et al. 87] Agrawal, R., Carey, M., & Livney, M. *Concurrency control performance modelling: Alternatives and implications.* ACM Transactions on Database Systems, Vol. 12, No. 4, pp. 609-654, December 1987.

[Aksit 96] Aksit, M. *Separation and Composition of Concerns.* ACM Computing Surveys 28A(4), December, 1996.

[Aksit 99] Aksit, M., Tekinerdogan, B., Marcelloni, F., & Bergmans, L. *Deriving Object-Oriented Frameworks from Domain Knowledge.* in: M. Fayad, D. Schmidt & R. Johnson (eds.), Building Application Frameworks: Object-Oriented Foundations of Framework Design, Wiley, 1999.

[Atkins & Coady 92] Atkins, M.S. & Coady, M.Y. *Adaptable Concurrency Control for Atomic Data Types.* ACM Transactions on Computer Systems, Vol. 10, No. 3, pp. 190-225, August 1992.

[Campbell et al. 93] Roy H. Campbell, Nayeem Islam, David Raila and Peter Madany; Commun. *Designing and implementing Choices: an object-oriented system in C++.* Communications of the ACM, Vol. 36, No. 9, pp. 117 - 126, September, 1993.

[Carey & Stonebraker 84] Carey, M., & Stonebraker, M. *The performance of concurrency control algorithms for database management systems.* In Proceedings of the 10th International Conference on Very Large Data Bases, Singapore, pp. 107-118, 1984.

[Chidamber & Kemerer 94] Chidamber, S.R. & Kemerer, C.F. *A Metrics Suite for Object-Oriented Design*. IEEE Transactions on Software Engineering 20(6): 476-93, 1994.

[Cross 89] Cross, N. *Engineering Design Methods*, Wiley and Sons, 1989.

[Fayad et al. 99] Fayad, M.E., Johnson, R.E., & Schmidt, D.C (Eds.). *Building Application Frameworks : Object-Oriented Foundations of Framework Design*. Wiley, 1999.

[Fenton & Pfleeger 97] Fenton, N.E., & Pfleeger, S.L. *Software metrics: a rigorous and practical approach.* London : International Thomson Computer Press, 1997.

[Jacobson et al. 99] Jacobson, I., Booch, G., & Rumbaugh, J., *The Unified Software Development Process*, Addison-Wesley, 1999.

[Jacobson et al. 92] Jacobson, I., Christerson, M., Jonsson, P., & G. Overgaard, Object-Oriented Software Engineering - A Use Case Driven Approach, Addison-Wesley / ACM Press, 1992.

[Jacobson 97] Jacobson, I., Griss, M., & Jonsson, P. *Software Reuse.* ACM Press, New York, 1997.

[Johnson & Foote 88] Johnson, R., & Foote, B. *Designing Reusable Classes*. Journal of Object-Oriented Programming. Vol. 1, No. 2, pp. 22-35, 1988.

[Kiczales et al. 97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, L., Loingtier, J.M., & Irwin, J. *Aspect-Oriented Programming*. ECOOP '97 Conference Proceedings, LNCS 1241, Springer-Verlag, pp. 220-242, 1997.

[Kumar 96] Kumar, V. *Performance of Concurrency Control Mechanisms in Centralized Database Systems*. Prentice-Hall, 1996.

[Lane 96] Lane, T.G. *Guidance for User Interface Architectures*. in: M. Shaw, D.Garlan: Software Architecture, Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[Lieberherr 96] Lieberherr, K.J. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[Nierstrasz & Tsichritzis 95] Nierstrasz, O. & Tsichritzis, D. (eds.), *Object-Oriented Software Composition*. Prentice Hall, 1995.

[Ossher & Tarr 99] Ossher, H., & Tarr, P. *Multi-Dimensional Separation of Concerns using Hyperspaces*. IBM Research Report 21452, April, 1999.

[Pree 94] Pree, W. *Design patterns for Object-Oriented Software Development*. Addison-Wesley, Reading, Mass, 1994.

[Riel 96] Riel, A.J. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[Roberts 96] Roberts, D., & Johnson, R. *Evolving Frameworks: A pattern language for Developing Object-Oriented Frameworks*. At URL: http://st-www.cs.uiuc/edu/users/droberts/evolve.html, 1996.

[Rumbaugh et al. 91] Rumbaugh,J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Schopbarteld 99] Schopbarteld, F. *Dynamically Tuning Transaction Behavior in a Distributed Environment*. MSc thesis, Dept. Of Computer Science, University of Twente, 1999.

[Saeki 98] Saeki, M. *Method Engineering*. in: P. Navrat and H. Ueno (Eds.), Knowledge-Based Software Engineering, IOS Press, 1998.

[Smith 90] Smith, C.U. *Performance Engineering of Software Systems*. Addison-Wesley, 1990.

[Steyaert et al. 96] Steyaert, P., Lucas, C., Mens, K., & D'Hondt, T. *Reuse Contracts: Managing the Evolution of Reusable Assets*. OOPSLA '96 Proceedings, ACM SIGPLAN Notices, pages 268-285, ACM Press, 1996.

[Taligent 96] Taligent white papers, at URL: http://www.taligent.com, 1996.

[Tekinerdogan 97] Tekinerdogan, B. & Aksit, M. *Adaptability in object-oriented software development: Workshop report*,in M. Muhlhauser (ed), Special issues in Object-Oriented Programming, Dpunkt, Heidelberg, 1997.

[Tekinerdogan & Aksit 99a] Tekinerdogan, B., & Aksit, M. *Providing automatic support for heuristic rules of methods*. In: Demeyer, S., & Bosch, J. (eds.), Object-Oriented Technology, ECOOP '98 Workshop Reader, LNCS 1543, Springer-Verlag, pp. 496-499, 1999.

[Tekinerdogan & Aksit 99b] Tekinerdogan, B., & Aksit, M. *Deriving design aspects from conceptual models*. In: Demeyer, S., & Bosch, J. (eds.), Object-Oriented Technology, ECOOP '98 Workshop Reader, LNCS 1543, Springer-Verlag, pp. 410-414, 1999.

[Wirfs-Brock et al. 90] Wirfs-Brock, R. Wilkerson, B., & Wiener, L. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

Index

Samenvatting

Sinds de introductie van de eerste programmeertalen in de jaren 1940 en 1950 heeft software engineering een aantal evolutionaire fases ondergaan die het mogelijk maakten om grootschalige en meer complexe software systemen te bouwen. Dit toegenomen potentieel werd spoedig gevolgd door het bewustzijn dat software zeer moeilijk af te leveren is binnen de beschikbare tijd, en budget en met de vereiste kwaliteitsfactoren zoals betrouwbaarheid, stabiliteit en aanpasbaarheid. Om een oplossing te kunnen bieden voor deze zogenoemde *software crisis* heeft men eind jaren 1960 het idee geopperd om een *engineering* aanpak binnen software ontwikkeling toe te passen.

Sindsdien zijn er binnen de software engineering discipline vele verscheidene pogingen ondernomen om de problemen die direct of indirect leiden tot de symptomen van de software crisis het hoofd te kunnen bieden. Een recente ontwikkeling is de toepassing van het software architectuur concept dat als fundamenteel middel wordt gezien voor het bouwen van kwalitatieve software systemen. Software architecturen representeren de gehele structuur van software systemen en hebben als zodanig een cruciale en directe invloed op de kwaliteitsaspecten van het systeem. Ondanks deze verschillende ontwikkelingen is het bouwen van kwalitatieve software systemen echter nog steeds een moeilijke taak en duren de symptomen van de software crisis voort.

Teneinde de essentie van software engineering en zijn problemen beter te begrijpen beschrijft dit proefschrift een grondig en kritische analyse van de software engineering discipline vanuit een breed perspectief. Daartoe wordt software engineering bekeken vanuit een probleem-oplossing perspectief waarbij software als oplossing wordt gezien voor de technische problemen. Software engineering wordt vervolgens vanuit een probleem-oplossingsmodel geanalyseerd en vergeleken met de meer gevorderde probleem-oplossingsgebieden zoals filosofie en de traditionele engineering disciplines zoals electrotechniek, civiele techniek, werktuigbouwkunde en chemische technologie. Uit deze analyse worden de lessen en concepten afgeleid die van belang kunnen zijn voor software engineering.

Een fundamenteel concept binnen gevorderde probleemoplossingstechnieken is het concept van *synthese* dat expliciete processen voor technische probleemanalyse, oplossingsdomein analyse en de analyse van alternatieven bevat en deze integreert. Binnen de software engineering is dit concept nagenoeg niet bekend en zijn deze drie processen niet goed geïntegreerd.

Dit proefschrift richt zich op de software-architectuur ontwikkelmethoden van software engineering. Teneinde deze beter te begrijpen wordt er een klassifikatie en evaluatie van de *state-of-the-art* methoden beschreven. Voor het oplossen van de problemen in deze methoden wordt er een nieuwe methode ingevoerd die gebaseerd is op het synthese-concept. Deze methode wordt geïllustreerd aan de hand van een voorbeeldproject uit de software-industrie waarbij een software-architectuur voor transactiesystemen wordt ontwikkeld.

Elke software architectuur kan gerealiseerd worden door middel van verschillende analyse- en ontwerpalternatieven. Het is belangrijk om de verschillende alternatieven expliciet weer te geven en deze af te wegen door middel van kwaliteitsfactoren, zodat geschikte alternatieven gevonden kunnen worden. Hiervoor introduceert het proefschrift een nieuw formalisme, genaamd *design algebra*. De technieken van design algebra kunnen geïntegreerd worden binnen de huidige object-georiënteerde software ontwikkelmethoden. Deze technieken worden automatisch ondersteund door een CASE tool, *Rumi,* dat speciaal hiervoor is ontwikkeld en in het proefschrift wordt beschreven.