Facilitating the Development of Design Support Software by Abstract Prototyping

Eliab Z. Opiyo
Facilitating the Development of Design Support Software by Abstract Prototyping
Facilitating the Development of Design Support Software by Abstract Prototyping

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. dr. ir. J. T. Fokkema,
voorzitter van het College voor Promotie,
in het openbaar te verdedigen

op dinsdag 21 januari 2003 te 16.00 uur

door

Eliah Zephania OPIYO

Master of Science in Mechanical Engineering, Cranfield University, United Kingdom
geboren te Ikizu, Bunda, Verenigde Repuliek van Tanzania
Dit proefschrift is goedgekeurd door de promotor:
Prof. Dr. I. Horváth

Toegevoegd promotor:
Dr. J. S. M. Vergeest

Samenstelling promotiecommissie:
Rector Magnificus voorzitter
Prof. Dr. I. Horváth Technische Universiteit Delft, Nederland
Dr. J. S. M. Vergeest Technische Universiteit Delft, Nederland
Prof. Dr. H. de Ridder Technische Universiteit Delft, Nederland
Prof. Dr. Ir. K. van der Werff Technische Universiteit Delft, Nederland
Prof. Dr. J. Vance Iowa State University, Verenigde Staten
Prof. Dr. T. Tomiyama Technische Universiteit Delft, Nederland
Prof. Dr. R. Rohatyński Technical University of Zielone, Polen
Prof. Ir. K. Robers Technische Universiteit Delft, Nederland (reserved)

Opiyo, E. Z.,
Facilitating the Development of Design Support Software by Abstract Prototyping,

ISBN 90-6734-136-3

Keywords: CAD/CAE/CAM, engineering software development, quality assurance.

Copyright © 2003 by E. Z. Opiyo. All rights reserved. No part of the materials protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without written permission from the author.
Preface

The increasing global competition, coupled with increased demands for product quality, reduced costs, shortened delivery time, and product flexibility have had a great impact on the product development and production process. As a result, concepts, methods, architectures and software tools used in product development frequently seem to be lacking in quality. It can be said that more powerful computer based product development systems need to be introduced continually. This thesis addresses the issue of assurance of quality and trustworthiness of software used in engineering design of artifactual products, referred to as design support software (DSS). The research work presented in this thesis was as an attempt directed towards narrowing the gulf between what is actually needed in quality assurance of DSS and what software engineering disciplines can offer. The research topic stemmed from the Integrated Concept Advancement (ICA) research program of the Department of Design Engineering, Delft University of Technology, which is concerned with the development of computer oriented theories and methods for product conceptualization.

In the process of development of DSS, the expected software reappears in various preliminary forms. It was hypothesized that the better the quality of the preliminary forms of implementations, the higher the chance of acceptance of the eventual DSS. Numerous concepts, models and methods for quality assurance of software are available. The current quality assurance practice is that phased software process models are applied in the development of DSS and tests are predominantly conducted after coding. Also, reviews are typically performed to remove faults before requirements or designs are passed to the subsequent phase. One of the problems is that the activities in the phases of the software process models, in particular in the design phase, are coarsely defined and do not scale to precisely match the needs in the processes of development of DSS. This research centered on this problem and involved the development of a theory and a computer based methodology to support developers in pre-implementation testing of DSS. This pre-implementation testing concept has been dubbed abstract prototyping (abbreviated as AP in this thesis). The specific activities have been to:

- review the past and on-going research activities to study: (i) quality issues in the development of DSS; and (ii) the existing testing techniques and related concepts;
- precisely define the AP concept, that is, to describe its meaning, its elements, its applications, its importance, its limitations, and so forth;
- develop a methodology and a suite of software tools to support AP activities; and
- study the validity of the AP concept.

The AP concept extends the current norms by introducing concrete milestones and deliverables at the design phase of the DSS development process. Based on this methodology, faults can as well be traced back to the in-process implementations of design process rather than exclusively to the requirements or final designs. Prototypes are used
to provide the feel or the look of the in-process implementations, and specifically designed metrics provide a means for estimation of the extent to which the implementations fulfill the stakeholders' demands. The principles of AP can be summarized as follows:

- The DSS design process passes through various different stages. There are various kinds of testable in-process implementations at the design phase of the DSS development process. These in-process implementations are referred to as the abstract prototypes, and the respective creation stages as the abstraction levels of the DSS.

- Requirements are clustered according to the abstraction levels. These requirements are the sources of the review criteria, and they can be improved through the elaboration processes that involve testing of the abstract prototypes.

- Representatives of various DSS stakeholders are systematically involved in the assessment of quality of the abstract prototypes as they evolve.

One of the primary goals of the AP concept is to involve the representatives of various DSS stakeholders in the development and review of the DSS at the design phase. Case studies have demonstrated that the AP concept adequately supports this endeavor. It provides the developers with a systematic way for finding flaws in solutions early on, based on views of the representatives of the DSS stakeholders. It thus helps to reduce the risk of developing unacceptable DSS.

This thesis gives an account of the research work. It first introduces the research problem and then presents a comprehensive review of the DSS development and testing process. Afterwards it describes the theoretical fundamentals of AP, the AP process and the pilot software implementation developed to support the developers of DSS in doing the AP activities. It also presents the real world application case studies conducted to explore the validity of the AP concept.
Acknowledgements

This research work has been done at the Department of Design Engineering, Faculty of Industrial Design Engineering, Delft University of Technology. The research topic is part of the research problems of the Integrated Concept Advancement (ICA) research program of the Design Engineering department. The author extends his sincere gratitude to his supervisor, Prof. Dr. Imre Horváth, and his co-supervisor, Dr. Joris S. M. Vergeest, for their guidance and contributions to this research work. He thanks them for the efforts and time they spent in reviewing the concepts of the research work, and in reading and correcting drafts. Their comments and suggestions have undoubtedly enhanced quality of this work. All members of the ICA research team have been incredibly supportive, and have contributed in one way or another in making this work a successful undertaking. The author expresses his appreciation for their valuable support. He also acknowledges the support he got from the Department and the Faculty managements, and he is indebted to Delft University of Technology for financing the research project.

Part of the research work took place at the Scheepsbouwkundig Advies en Reken Centrum (SARC) BV of Bussum, the Netherlands. The author gratefully thanks Dr. Herbert Koelman, the founder and the CEO of SARC who made this possible, and Mr. Paul Zevenbergen for his valuable day-to-day support.

Finally, the author is indebted to all members of the promotion committee, who reviewed the initial draft of the thesis. Their valuable comments and suggestions have improved this thesis to its present form.

Eliab Zephania Opiyo
To Rest, Andrew, Anelh.

Without your encouragement and support, the research documented in this thesis would not have been completed.
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM</td>
<td>Association for Computing Machinery, Inc</td>
</tr>
<tr>
<td>AP</td>
<td>Abstract Prototyping</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>APN</td>
<td>Advanced Petri-nets</td>
</tr>
<tr>
<td>B Rep</td>
<td>Boundary Representation</td>
</tr>
<tr>
<td>CACD</td>
<td>Computer Aided Conceptual Design</td>
</tr>
<tr>
<td>CAD</td>
<td>Computer Aided Design</td>
</tr>
<tr>
<td>CAD/CAM</td>
<td>Computer Aided Design/Computer Aided Manufacturing</td>
</tr>
<tr>
<td>CADE</td>
<td>Computer Aided Design and Engineering</td>
</tr>
<tr>
<td>CAE</td>
<td>Computer Aided Engineering</td>
</tr>
<tr>
<td>CAPP</td>
<td>Computer Aided Process Planning</td>
</tr>
<tr>
<td>CEO</td>
<td>Chief Executive Officer</td>
</tr>
<tr>
<td>CGAL</td>
<td>Computational Geometric Algorithms Library</td>
</tr>
<tr>
<td>CIM</td>
<td>Computer Integrated Manufacturing</td>
</tr>
<tr>
<td>CLIPS</td>
<td>C Language Integrated Production System</td>
</tr>
<tr>
<td>CMM</td>
<td>Capability Maturity Model</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DNA</td>
<td>Deoxyribonucleic Acid</td>
</tr>
<tr>
<td>DoF</td>
<td>Degrees of Freedom</td>
</tr>
<tr>
<td>DSS</td>
<td>Design Support Software</td>
</tr>
<tr>
<td>GL</td>
<td>Graphics Library</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HIPO</td>
<td>Hierarchy-Process-Input-Output</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines</td>
</tr>
<tr>
<td>ICA</td>
<td>Integrated Concept Advancement</td>
</tr>
<tr>
<td>IDC</td>
<td>Internet Database Connector</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electro technical Commissions</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>IGES</td>
<td>Initial Graphic Exchange Specification</td>
</tr>
<tr>
<td>ISO</td>
<td>International Standards Organization</td>
</tr>
<tr>
<td>KRP</td>
<td>Knowledge Representation and Processing</td>
</tr>
<tr>
<td>MDI</td>
<td>Multi-Document Interface</td>
</tr>
<tr>
<td>MDKS</td>
<td>Multi Dimensional Knowledge Search</td>
</tr>
<tr>
<td>MDRS</td>
<td>Multi Dimensional Requirements Search</td>
</tr>
<tr>
<td>MFC</td>
<td>Microsoft Foundation Class</td>
</tr>
<tr>
<td>MSF</td>
<td>Microsoft Solutions Framework</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>NC</td>
<td>Numerical Control</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>NURBS</td>
<td>Non Uniform Rational B-Splines</td>
</tr>
<tr>
<td>ODBC</td>
<td>Open Database Connectivity</td>
</tr>
<tr>
<td>OLE</td>
<td>Object Linking and Embedding</td>
</tr>
<tr>
<td>OO</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>OOAD</td>
<td>Object Oriented Analysis and Design</td>
</tr>
<tr>
<td>OpenGL</td>
<td>Open Graphics Language</td>
</tr>
<tr>
<td>PAS</td>
<td>Publicly Available Specifications</td>
</tr>
<tr>
<td>Q&amp;A</td>
<td>Question and Answer</td>
</tr>
<tr>
<td>R&amp;D</td>
<td>Research and Development</td>
</tr>
<tr>
<td>RRP</td>
<td>Requirements Representation and Processing</td>
</tr>
<tr>
<td>RTF</td>
<td>Rich Text Format</td>
</tr>
<tr>
<td>SPM</td>
<td>Software Process Management</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>STEP</td>
<td>Standard for the Exchange of Product Model Data</td>
</tr>
<tr>
<td>TRIZ</td>
<td>Teorija Reszenia Izobretatelnyh Zadatch - (Russian acronym for Theory of Inventive Problem Solving)</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>UML</td>
<td>Universal Markup Language</td>
</tr>
<tr>
<td>URL</td>
<td>Universal Resource Locator</td>
</tr>
<tr>
<td>V&amp;V</td>
<td>Verification and Validation</td>
</tr>
<tr>
<td>VDIM</td>
<td>Vague Discrete Interval Modeling</td>
</tr>
<tr>
<td>VDM</td>
<td>Vienna Development Method</td>
</tr>
<tr>
<td>WIMP</td>
<td>Windows-Icons-Menu-Pointer</td>
</tr>
</tbody>
</table>
# Table of Contents

<table>
<thead>
<tr>
<th>Preface</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>vii</td>
</tr>
</tbody>
</table>

## 1. Introduction

1.1 Background  
1.2 Supporting Design by Computers  
1.2.1 Development of Design Support Software Systems  
1.2.2 Assurance of Quality of the Design Support Software  
1.2.3 Overview of the Primary Research Program  
1.3 Problem Definition  
1.4 Research Hypothesis  
1.5 Research Objectives and Activities  
1.6 Outline of the Thesis  
1.7 Publications  

## 2. Quality Assurance of Design Support Software: A Literature  

2.1 Background  
2.2 Analysis of the Processes of Development of Design Support Software  
2.3 Assuring the Process of Development of Software  
2.3.1 Software Development Models  
2.3.2 Automated Software Production  
2.3.3 Software Process Management Concepts  
2.4 Software Verification and Validation  
2.4.1 Software Testing  
2.4.1.1 Classical Tests  
2.4.1.2 Review Methods  
2.4.2 Controlling the Levels and Amounts of Details  
2.4.2.1 Abstraction  
2.4.2.2 Software Prototyping  
2.5 Other Software Quality Assurance Issues and Strategies  
2.5.1 Requirements Engineering  
2.5.2 User Centered Quality Assurance  
2.5.3 Knowledge of the Developers  
2.5.4 Software Quality Standards  
2.5.5 Metrics, Measurements and Consensus Making  
2.6 Summary  
2.7 Discussion and Concluding on a Concept for Pre-Implementation Testing of Design Support Software  
2.8 Conclusion
3. **Theoretical Fundamentals of Abstract Prototyping**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Setting the Abstract Prototyping Reference Model</td>
<td>43</td>
</tr>
<tr>
<td>3.2</td>
<td>Basic Concepts and Terms</td>
<td>46</td>
</tr>
<tr>
<td>3.3</td>
<td>Abstraction Levels</td>
<td>49</td>
</tr>
<tr>
<td>3.4</td>
<td>I/O of Abstract Prototyping</td>
<td>51</td>
</tr>
<tr>
<td>3.5</td>
<td>Role of Users and Developers</td>
<td>51</td>
</tr>
<tr>
<td>3.6</td>
<td>Requirements Engineering in the Perspective of the Abstract Prototyping Concept</td>
<td>53</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Fundamentals</td>
<td>54</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Acquisition of Requirements</td>
<td>59</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Requirements Model</td>
<td>59</td>
</tr>
<tr>
<td>3.7</td>
<td>Knowledge Management in Abstract Prototyping</td>
<td>61</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Representation of Knowledge</td>
<td>62</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Relationships among Abstract Prototypes</td>
<td>62</td>
</tr>
<tr>
<td>3.8</td>
<td>Conclusion</td>
<td>64</td>
</tr>
</tbody>
</table>

4. **Abstract Prototyping Process**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Application Methodology</td>
<td>65</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Abstract Prototyping Activities</td>
<td>65</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Driving Schemes</td>
<td>69</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Opinions Gathering</td>
<td>70</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Application Scenarios</td>
<td>71</td>
</tr>
<tr>
<td>4.2</td>
<td>Computer Based Activities</td>
<td>72</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Requirements Representation and Processing</td>
<td>73</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Knowledge Acquisition, Representation and Processing</td>
<td>75</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Information Analysis</td>
<td>78</td>
</tr>
<tr>
<td>4.3</td>
<td>Conclusion</td>
<td>80</td>
</tr>
</tbody>
</table>

5. **Design and Implementation of the Abstract Prototyping Software**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Needs Analysis and Allocation of Features to the Abstract Prototyping Software</td>
<td>81</td>
</tr>
<tr>
<td>5.2</td>
<td>Requirements for the Abstract Prototyping Software</td>
<td>82</td>
</tr>
<tr>
<td>5.3</td>
<td>Design of the Abstract Prototyping Software</td>
<td>84</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Architectural Design</td>
<td>85</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Information Flow</td>
<td>87</td>
</tr>
<tr>
<td>5.4</td>
<td>Implementation</td>
<td>88</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Requirements Representation and Processing Module</td>
<td>89</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Knowledge Representation and Processing Module</td>
<td>91</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Opinions Gathering Module</td>
<td>93</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Information Analysis Module</td>
<td>95</td>
</tr>
<tr>
<td>5.4.5</td>
<td>Warehouse</td>
<td>99</td>
</tr>
<tr>
<td>5.4.6</td>
<td>Auxiliary Utilities</td>
<td>102</td>
</tr>
<tr>
<td>5.4.7</td>
<td>User Interfaces</td>
<td>104</td>
</tr>
<tr>
<td>5.5</td>
<td>Conclusion</td>
<td>105</td>
</tr>
</tbody>
</table>
Introduction

This chapter introduces the research work. It first describes the background of the research and the nature of the problem, and then presents the research hypothesis, objectives and activities. It finally outlines the structure of the thesis.

1.1 Background

Many researchers and practitioners regard design as the most influential activity in the product development process. Product development activities can broadly be categorized as: (i) obtaining market requirements and clarifying the problem; (ii) generating product ideas and concepts, and subsequently formalizing the specifications, which help orient design and manufacturing; (iii) transforming the specifications into a concrete product; and (iv) launching the product destined to satisfy previously specified market needs and expectations. Many interconnected sub-processes, both internal and external to the company, constitute the product development process. Such sub-processes include marketing, conceptual design, embodiment design, detailed design, process planning, production planning and manufacturing. Figure 1.1 shows the interrelationships among various product development sub-processes and the available computer support.

Changes in the market requirements as well as the emergence of new product development concepts and computing technologies have had a great impact on the product development processes. Due to the increased or frequent change of requirements, some of the architectures and software tools applied in product development seem to be inadequate. The industrial companies face common challenges, namely, how to: (a) meet the market requirements for product quality; (b) reduce the development and production costs; (c) shorten time-to-market; and (d) enhance product flexibility while improving productivity. Several computer-based design and manufacturing concepts have been created as attempts to deal with this challenge, and many experimental software solutions are emerging. Today, numerous Computer-Aided X (CAX) systems commonly known as Computer-Aided Design and Manufacturing (CAD/CAM), Computer Aided Process Planning (CAPP) and Computer Integrated Manufacturing (CIM) are available in various different names and versions.

1 The term 'product' is used in this thesis to refer to artifacts or services produced or provided through industrial, mechanical, or software engineering processes.
Many publications and new research initiatives in recent times have addressed the need for either enhancement of some of the existing computer based methods or development of new architectures for some product development activities [see e.g. Tovey, (2000); Slingeland, (2000)]. Researchers and practitioners have frequently expressed concerns about the need for improvement of the existing methods and software tools to accommodate new requirements. The need for extending computer support to include activities that are traditionally not supported by computers is also emphasized. It is pointed out that the increasing competition and changes in the markets have been the reason for the need for more effective methods and software tools. The researchers and developers have one primary challenge in common, namely, how to create suitable computer-oriented methods and software tools. What product development and production proc-
esses need is quality methods and software tools that can help achieve, among other things, reduced product development and production costs, shorter time-to-market, and improvement of quality of products. Clearly, concepts used in the development and testing of computer-oriented methods and software tools used in product development have direct influence on quality of the eventual methods or software products.

Currently there is a plethora of computer-based methods and tools for supporting product development and production in the academia and research institutions. While novel methods and software tools are mushrooming, the proportion introduced in the industry is rather disappointing [refer to, for example, Birkhofer et al. (2002); Medland and Mullineux (2002); Rohatynski (2001); Vergeest et al. (2001)]. On the other hand, the rate of arrival of new versions of existing software in the markets is generally high. This is partly due to the fact that most often these software tools do not live up to the expectations. It can be said that quality assurance of the computer-based methods used in the development of products need special emphasis from both the academia and engineering software development firms. The organizations entrusted to develop these software tools need to direct their efforts towards improving the reliability and quality for their tools to meet the highest expectations. After all, developing reliable and high quality tools is not a fad for them, but a requirement to stay in business and to maintain competitiveness.

1.2 Supporting Design by Computers

Many professional and scientific articles advocate that design is a pivotal process in product development such that its improvement is crucial to any effort aimed to achieve industrial excellence and to increase competitiveness [refer e.g. to Duffy (2002); Sharpe (1995); Szafarczyk (2002)]. In recent times, remarkable steps have been made towards partial automation of the design process through the use of dedicated software systems. Numerous design support software systems, variously known as Computer-Aided Design (CAD) systems, Computer-Aided Design and Manufacturing (CAD/CAM) systems, or Computer-Aided Engineering (CAE) systems have been developed and they offer a large spectrum of capabilities and support in the product design processes. The term Design Support Software (or acronym DSS) is used in this thesis to refer to any software tool used either in the conceptualization, the detailed design, or in the engineering-analysis stage of the product development process.

The significance of quality assurance to DSS is pretty comprehensible. Using DSS, for instance, in the design and analysis of products or systems is critical for both safety and property protection. If, for example, DSS used in the analysis or prediction of what will happen to systems or components of systems such as an automobile or an airplane when put into service produces incorrect results, the consequence could be a failure that could ultimately lead to loss of life, property or environmental damage. It is therefore
crucially important to ensure quality of DSS\(^2\) before they are put in service even if this adds to their development time and costs.

Publications, however, indicate significant limitations in terms of functionality, interaction with, and reliability in most of the prevailing DSS systems [see e.g. the NIST report, 2002]. For instance, the conventional CAD systems are typically useful only in the final stages of the design process where they are used mainly for documentation of the results of mental design activities, as processors for generation of product models needed in engineering analysis, and as front-ends for NC programming. In the past decades, conceptual design, which is perhaps the most critical stage of the product design process in terms of costs, performance and quality of products, received little attention from researchers, from the point of view of computer support\(^3\) [Horváth, 1998]. The situation has somewhat changed in recent times, and various researchers worldwide are now focusing on this very challenging issue. However, it is proclaimed in the literature that application of computers in conceptual design is still minimal [refer, for instance, Dijk et al., (1998); van Elsas and Vergeest, (1998); Hennessey, (1993)]. Some of the software tools presently used in conceptual design have simply been adapted from specific fields such as surface design, or system analysis and simulation. The common understanding and experience among researchers and practitioners is that conceptual design is very difficult to computerize [refer to, for instance Dijk, 1995]. To a large extent, this is attributed to, among other things, the lack of robust theoretical bases, lack of common methodology, vagueness of design information at the early phases, and difficulty of handling various design aspects (such as aesthetics, manufacturability and ergonomics) simultaneously and objectively. Another example on the deficiencies in the existing DSS systems is the limited ability of graphical user interfaces to support interaction during design. Nowadays, designers predominantly use Windows-Icons-Menus-Pointer (WIMP) interfaces to interact, for instance, with shape models. Nevertheless, there is strong evidence in the literature that the WIMP interface does not offer effective interaction and adequate support of modeling in conceptual design [Dani et al., (1997); Horváth, (1998); Tovey, (1997)]. There is a need of developing new interaction methods that will take the advantage of the capabilities of the emerging input techniques, such as those based on speech, hand motion, and gesture, which will more effectively match human practices and capabilities. There is also evidence in the literature of various DSS being developed to support designers in carrying out specific activities, but failed to be as useful as expected due to a lack of various desirable quality or functional characteristics [see e.g. Tay and Gu, 2002]. These kinds of limitations not only affect the designers and the design process, but also affect the quality of the designs.

\(^2\) Quality of DSS in this case refers to among other quality characteristics, the quality of data generated by the DSS. Quality of interpretation of the data generated by the DSS is also equally crucial.

\(^3\) However, it should be noted that some studies show that some designers still prefer performing conceptual design activities manually rather than by using computers.
It can be said that the DSS: like other software products, are neither necessarily completely free from faults nor do they always precisely live up to the expectations. The ever-changing needs, the advances in computing technology, and the constantly evolving technological infrastructure of the modern world continually bring about new functional and quality requirements. Like for other software products [refer, for instance to Lieberman and Fry, (2001)], it is difficult to fully satisfy all needs of the designers. This is attributed, to some extent, to the limitations of the existing software development models and quality assurance strategies, which do not precisely match the needs of the process of development of DSS.

1.2.1 Development of Design Support Software Systems

Phased models applied in software engineering are typically taken for granted and applied in the development of DSS. Most widely used development models such as the waterfall, program growth, and rapid prototyping, include requirements analysis and specification, design, implementation, testing and installation, and operation as the main phases of software development[^4]. One of the goals of phased development is to minimize faults in the delivered codes. Tests are typically conducted at the 'implementation' phase, at which parts of the programs are evaluated as they are completed, and at the 'testing and installation' phase, where all parts of the software are put together and evaluated [Bashir and Goel, (2000); Jones, (1990)]. On the other hand, reviews are typically performed in order to remove faults before passing the requirements or designs to the subsequent phase. The problem is that activities in the phases of software process models, in particular in the design phase[^5], are coarsely defined and that they do not scale to precisely match the needs in the development of DSS. Most software tools are built for very pragmatic reasons and based on known knowledge [Amour, 2001 b], rather than starting from scratch, by developing original foundational theories. During the development interval, the developers work on the software, which appears in three different forms, namely, as specifications, then as a design, and finally as codes. Faults in codes can typically be traced back to the requirements or designs.

In principle, the processes of development are similar for all types of software products[^6]. However, in contrast, the DSS products are different from other software in that they are based on engineering principles or physical phenomena [refer, for example, to Kreyman et al., (1999) and Zeid, (1991)]. Empirical evidences as well as the literature show that the design phase of the DSS development process is broad and rich in research, and that the developers work with the envisaged DSS in various different guises, reflecting abstraction of DSS in various contexts [see, for example, Dijk et al., 1998]. Furthermore, as opposed to other software, the DSS are typically complex, and it is difficult to comprehensively formulate their requirements when developing them in

[^4]: Section 2.3.1 exclusively reviews the process models used in software development.

[^5]: The term 'design phase' refers to both software conceptualization activities (such as ideation of working principles of software, user interfaces, etc.) and detailed design activities (i.e. design of algorithms).

[^6]: There are various kinds of software. They can be classified differently, for instance, according to their application domains, services they offer to users, and so forth.
the frameworks of the conventional software development models. It can be said that
the design phase of the DSS needs to be described in a more granulated way, and vari-
ous milestones and abstraction levels of a DSS must be formalized. Ensuring quality of
the DSS in various contexts at the design phase can warrant acceptance of the eventual
DSS.

1.2.2 Assurance of Quality of the Design Support Software
Software is increasingly becoming a crucial tool in the design of many of today's prod-
ucts. There is an ever-growing demand on quality of DSS and the product design com-
panies are confronted by, among other things, occasional introduction of completely
new applications or new versions of existing applications as well as changes of envi-
ronments, methodologies and/or platforms. Software products used in many profes-
sions, businesses, or for providing services to the public (for example web-based appli-
cations) have long been a concern in the software engineering community, and there has
been a great deal of research work on quality assurance of these software. Ironically,
despite their significance, software packages used at the design phase to support model-
ing and analysis of engineering or consumer products have not received the same level
of attention [Kreyman et al., 1999]. It can generally be said that the DSS developers'
community has been deprived of appropriate tools or methods for shaping the develop-
ment activities, and for quality assurance.

Quality of a DSS is judged based on how well the software satisfies the designers'
needs, typically stated explicitly or implicitly in the requirements model. Quality assur-
ance is primarily the responsibility of the developers. The traditional software engineer-
ing strategies for requirements definition, design, implementation and testing are typi-
cally followed to assure quality in the development of DSS. There are several classical
software development models and testing procedures in place. Norms such as adherence
to systematic development procedures, following good programming styles, and use of
safer programming languages are observed, and various verification and validation
techniques are applied in companies to assure quality. In some cases, the developers
also use non-traditional methods in order to, for example, get a quick impression on the
user acceptance, and so forth. Nevertheless, complaints about the inadequacy of the
DSS are still common.

Despite the presence of numerous strategies for quality assurance, there are still signifi-
cant pitfalls, and occasionally faults pass through the development processes, ending up
as bugs in the delivered software. In the DSS development projects (and in other soft-
ware development projects as well), software project failure is common, and often the
users hold the developers liable for the failures. Various studies, see for instance, the
Standish Group's Chaos Report (1994), show that many software development efforts
do not succeed in delivering good quality products in time and within budgets, and that
the developers frequently experience great difficulties in satisfying the needs of the
end-users. Often it is not until late phases of the development process, or even as late as
after delivery, that the end-users can conclude that the purchased software package does
not meet their expectations [NIST Report, (2002); Dekkers, (2000)]. These problems
can partly be attributed to (i) usage of inadequate development models, (ii) adoption of inadequate verification and validation procedures; or (iii) fuzzy understanding of the requirements. Also, part of the problem is that many of the present testing procedures typically require partially or fully implemented software prototypes. As a result, the verification and validation processes can become lengthy and expensive, especially if a sufficient solution is not achieved in the first trials. Experiences in areas such as human computer interaction (HCI) show that conducting reviews early on, prior to coding, can significantly help assure quality, and may also shorten the development time and reduce the development costs [refer to, e.g. Casaday and Cynthia (1995); Bayer and Holtzblatt (1998)].

Quality assurance, as a discipline, is rooted in manufacturing [Beizer, 2001 a]. Evidently, there are differences between software products and mechanical engineering or consumer products. For example, often the product developers in industrial or mechanical engineering firms have preconceived notions about the products. This is not typical in software development. Similarly, software, and indeed software projects differ significantly. There is a common understanding in some circles of the software engineering community of the need for refined development models and/or verification and validation procedures, specific for different categories of software products. As for using the traditional quality assurance strategies in the development of DSS, the main problem is that they do not provide an adequate framework for development and reviewing or testing all in-process implementations systematically, particularly at the design phase. For example, when working in the framework of the existing models, theories, which form foundations for functions are neither considered as one of the guises of a DSS, nor formally reviewed. Similarly, the underlying methods and algorithms are typically not reviewed formally. Another problem is that the application of some techniques such as automated software production procedures, tend to distance the expertise of the developers and exclude experiences of the end-users from the test floors while others, for instance the usability tests, overwhelmingly lead to testing of specific users' requirements [Whiteside et al., 1988]. Like for other software products, ensuring that both the stakeholders and the infrastructure requirements are taken into consideration sufficiently and harmoniously is a big challenge for the developers of DSS as well.

In conclusion, it can be said that the traditional software development models can be used successfully as meta-models for guiding the DSS development activities and quality assurance. However, to effectively support the DSS development process, the activities within the phases need to be defined finely. Reviews need to be introduced and interwoven into the DSS design process, which according to the literature [refer, for example, to Eick et al., 1992] accounts for significant number of faults. It is also important to involve representatives of various DSS stakeholders in the reviews. Furthermore, validation of requirements\(^7\) needs to be given due consideration. In the processes of development of DSS, it often turns out that "obvious" requirements prove to be ob-

\(^7\) Validation of requirements basically involves investigation of whether the requirements have been correctly chosen or assumed, and if they can be fulfilled.
scure or redundant. For example, in Van Dijk (1994), seemingly tangible and indisputable requirements such as 'producing high-quality pictures during 2D or 3D sketching' turned out to be counter-productive, whereas seemingly intangible requirements like 'small cracks in a geometric model must be avoided' turned out to be substantial later on. It is therefore important to explicitly investigate the validity of requirements.

1.2.3 Overview of the Primary Research Program

This work is part of the Integrated Concept Advancement (ICA) research program, in which the global objective is to develop computer-based theories and methods for product conceptualization [refer to Section E.2 in Appendix E for further details]. The scope of conceptual design is generally not agreed upon among researchers and practitioners. There is a general consensus, however, that conceptual design is the early stage of the product design process that follows after market and feasibility studies. The outputs of conceptual design are design solutions, often in the form of conceptual shapes or non-shape information that may be used as a basis for embodiment and detail design. The process of generating initial, or in other words, conceptual shapes in conceptual design is called shape conceptualization [Figure 1.2]. It consists of the following activities: (i) formation of mental ideas about the intended product, (ii) expressing images, (iii) synthesis of separate ideas into a complete whole solution, (iv) generating an initial representation, and (v) physical concept modeling.

![Shape Conceptualization Process Diagram](image)

**Figure 1.2:** Shape conceptualization process

The goal of computer support of the shape conceptualization process is to facilitate the transformation of mental images into handy and formal graphical shape models, which present interim principal ideas about the expected appearance and operation of the
product. A number of design aspects such as ergonomics, aesthetics, cost, manufacturability and many others are typically taken into account.

Objectives of the Primary Research

The ICA research program aims at providing designers with: (i) techniques for communicating shape information to computers in a natural and a highly interactive way [Kuczogi et al., 2001]; (ii) modeling techniques that can cope with inconsistencies and contradicting input of shape information [Rusák et al., 2000 a]; (iii) procedures for preparation of manufacturing information for production of large-sized physical concept models using available manufacturing technologies [Horváth et al., 2000]; and (iv) strategies to facilitate handling of knowledge related to various design aspects such as aesthetics, cost, and manufacturability; based on ontology and nucleus paradigms [Pulles et al., (1999); Jambak et al., (2002)]. Several research issues are dealt with, including: (i) how to facilitate knowledge capturing and formalization; (ii) how to arrange the acquired information into meaningful knowledge units; (iii) how to apply the existing and emerging interaction techniques such as hand motion, speech and spatial scanning to communicate knowledge on shape intent; (iv) how to treat the sequence and contents of input knowledge; and (v) how to handle vagueness of input shape information. Traditionally, designers use natural techniques such as combinations of hand gestures and verbal utterances, two-dimensional sketches and claying in expressing concepts. The intention in this research program is to study the potentials of adopting these methods. While the research aims at maintaining the intuitiveness and the flexibility that designers enjoy when working in the frameworks of the traditional methods, the goal is also to achieve improvement in the efficiency and productivity, and to ensure usability of the outputs of the conceptual design activities. It is desirable that shape information must be represented in a form that can be used in the subsequent design stages of embodiment and detailed design. Thus, part of the research work deals with the challenges of: (i) how to represent conceptual shapes modeled by using natural interaction means; and (ii) how to convert natural shape descriptions into the standard representation formats. Prototype software tools to demonstrate various shape conceptualization concepts are being developed, and will be integrated in a prototype Computer Aided Conceptual Design (CACD) system.

Research Methods and Links of this Work to the Main Research Program

The development of various utilities of the envisaged CACD system involves three main tasks, namely, exploratory study, experimentation with prototypes, and implementation. The exploratory work is primarily concerned with finding possible solutions using methods such as literature surveys, interviews and brainstorming. The end products of this activity are theoretical (conceptual) or methodological solutions. A function of software can be built based on several theoretical or methodological alternative solutions. There can also be alternative algorithms or existing codes that match the problem at hand. The ultimate goal is to come up with a consistent, complete and usable prototype CACD system. One way of achieving this is through experimentation with prototypes in the early phases so that faults in the conceptual solutions, which form founda-
tions for the implementation of the CACD system, can be detected and eliminated early on, before codes are written.

The work reported in this thesis is aimed at providing the developers of the DSS with means to systematically ensure quality of the in-process implementations, and to eliminate faults prior to coding. To achieve this, the work was intended to provide a framework for acquiring and maintaining the understanding of requirements in the early stages of the DSS development. Furthermore, one of the objectives was to stimulate the understanding and communication between the developers and the stakeholders in the early phases of the DSS development process. By default, this could also help the developers to reconcile the requirements of various DSS stakeholders, and at the same time, facilitate a smooth introduction of novel computer based methods into the industry.

1.3 Problem Definition

The problem dealt with in this research relates to quality assurance of the DSS. The specific research question was how to support early phases of DSS development, and how to foster the discovery and elimination of faults in the DSS prior to coding. The main problem is that the prevailing software process models and testing strategies cannot adequately meet the needs of DSS development projects. In particular, the granularity of the design process steps is too large, and do not conform to the actual needs. The present quality assurance practice is that software process models and testing strategies are applied to ensure compliance of DSS with desirable quality characteristics. These models are typically multi-phased, and consist of requirements specification, design, implementation, testing and operation as the main phases. Many faults can be traced back to the pre-implementation phases of requirements specification and design, and reviews are typically conducted to remove faults in these phases. However, most of the fault detection techniques currently used in the early phases tend to focus on limited quality attributes, for example, assurance of usability of DSS, while other key desirable characteristics such as accuracy and completeness are not given due considerations. In general terms, the existing quality assurance schemes are porous or deficient. Part of the problem is that the activities in the software process models, in particular in the design phase, are coarsely defined, and do not scale to precisely match the underlying needs. As a result, significant numbers of faults originate in this phase and pass to the implementation phase. Furthermore, the design phase activities, which are to some extent operatively research oriented in the sense that they involve exploration or adaptation of foundational theories and methods, proceed in an ad hoc manner. There are typically no formal in-process milestones, and the in-process implementations are traditionally not reviewed or tested formally. The end product of such a disorganized process is most likely to be susceptible to flaws.

The inadequacy of the software process models and testing techniques is also attributed to the fact that the DSS are somewhat different from other software products. Unlike the software used in other professions, the DSS are based on engineering principles or
physical phenomena. Also, they are typically complex software tools, and it is difficult to adequately specify and address their requirements when working in the framework of the conventional software development models. Another research issue is that currently most DSS development projects are being based on technologies or standards from few leading organization or companies. This means that in most cases, the theories or the concepts underlying the DSS tend to remain intact regardless of the new requirements. Therefore, there is always the danger of not addressing sufficiently, e.g., the end-users needs. These issues pose a unique quality assurance challenge. This research work dealt with the above-mentioned problems, and was concerned with development of a theory and a computer based methodology for pre-implementation testing of DSS.

1.4 Research Hypothesis

Traditionally, DSS development involves, among other things, specification of requirements for the final DSS product, and testing predominantly after coding. The hypothesis of this research is: there must be a mechanism that can support specification of requirements for the DSS as it resurfaces in various appearances in the design phase and that can allow for reviews to be conducted systematically prior to coding. It is hypothesized that the process of designing a DSS passes through various stages, implicating distinct abstraction levels and contexts, and yields various different in-process implementations. A formal methodology can be created and used to systemize the process of designing DSS and testing the in-process implementations. Designing DSS by treating the in-process implementations as tangible and testable products allows for consideration of many additional requirements, and this can improve the understanding of the problem and give a clearer picture of what the eventual DSS should be. Furthermore, it is hypothesized various stakeholders can systematically be involved in reviewing the in-process implementations in an orderly manner. The imagined review methodology can be flexible and phased, so as to facilitate containment of faults close to their sources. This methodology should be applicable in diverse DSS projects, facilitate handling of emerging requirements, and help reconcile the needs of various DSS stakeholders. Application of such a methodology can improve quality and trustworthiness of the DSS, and ultimately warrant its acceptance. The hypothesis can be tested by applying the envisaged methodology in some selected DSS development projects, where, for instance, its influence on the number of requirements can be quantified, and in so doing its validity verified. Representatives of future users can also be asked to indicate their acceptance of the envisaged methodology as a quality assurance strategy.

---

8 For example, currently the visualization functions of some CAD/CAM/CAE systems are based on OpenGL, a Silicon Graphics, Inc. product. The same graphics environment [OpenGL] is used by developers in diverse markets such as broadcasting, entertainment, medical imaging, and virtual reality. It might not be true in this case, but some software engineering scholars [see e.g. Glass, 2002 b] argue that such problem-independent-solutions tend to be weak.
1.5 Research Objectives and Activities

The activities in this research work have been threefold, namely: (a) development of a theory and a methodology for pre-implementation testing of DSS; (b) development of software tools to support the developers and testers in pre-implementation testing of DSS; and (c) conducting case studies to scrutinize the validity of the pre-implementation testing concept. The focus of the research has been on studying issues such as: (i) how can quality and trustworthiness of DSS be assured prior to coding, (ii) how to pursue this objectively, (iii) how to check systematically whether the users and the infrastructure requirements are being taken on board as the DSS design progresses, and (iv) how to systematically involve various stakeholders in the DSS design process.

The specific objectives have been to:

- Review the state of the art and practice, as well as past and on-going research activities to: (i) study quality issues in the development of DSS, and (ii) analyze the existing software development models, software verification and validation techniques, and other related concepts.
- Lay down the principles for pre-implementation testing of DSS so as to allow for the in-process products in the design interval to be systematically reviewed and faults detected early on.
- Precisely define a formal concept for pre-implementation testing of DSS, namely, to clearly state what it means, and elaborate on its application, importance, and limitations.
- Develop a formal method and software tools for pre-implementation testing of DSS. As a formal method, the pre-implementation testing methodology, among other things, has to address pragmatic considerations such as who uses it, what it is used for, when it is used, and how it is used [Wing, (2000 b); Young, (1991)]. The pre-implementation testing activities may or may not have software tools to support them. The methodology should possess a set of guidelines that tell the users the circumstances under which the methodology can and should be applied, as well as how it can be applied most effectively.

Therefore, the tasks involved included:

- systemization of the DSS pre-implementation testing process, formalization of its elements, and identification of manual and computer supported activities;
- specification of requirements for a software system for supporting the DSS developers in pre-implementation testing, that is, to clearly and precisely describe the required functions, their performance constraints and their acceptance criteria;
- designing the software; and
Implementation and testing of software.

- Investigate the usability of the pre-implementation testing concept.

As an attempt to deal with the challenges presented in Sections 1.2 and 1.3, the abstract prototyping (AP) concept has been introduced to support developers in pre-implementation testing of DSS. A proof-of-concept suite of software tools for supporting the pre-implementation testing activities has also been developed and tested. The exclusive feature of the AP concept is that it puts into the spotlight the design phase of the DSS development process and finely defines its stages and activities. It can be said that it provides a framework for shaping the development activities at the design phase and for pre-implementation testing of the DSS. It also provides a scheme for involving the representatives of various stakeholders in reviewing the DSS during their design. The idea is to provide the developers of DSS with appropriate support in the form of a theory and a methodology for detecting and eliminating faults early on, as well as a computer-based information system to boost their competences and capabilities in this pursuit.

1.6 Outline of the Thesis

This thesis presents the research work, and it is structured as follows. The following chapter gives a comprehensive review of the state-of-the-art of the strategies used in the development and testing of software, including DSS. Various concepts and techniques used in the development and in verification and validation of DSS as well as other software products are presented and discussed. Chapter 3 introduces the theoretical fundamentals of AP. The AP reference model, terminology and the basic concepts are introduced. Chapter 4 describes the AP process. The AP activities and various higher-level procedures are concretely defined and described. Chapter 5 presents the prototype software tools developed to support the DSS developers in doing the AP activities. The underlying algorithms and key functions are described. Chapter 6 first outlines the guidelines of using the AP software. Then, it describes the real world application case studies of the AP methodology and software, and presents the main findings. Chapter 7 concisely summarizes the research work and discusses the results of the research work, presents conclusions, and recommends possible directions of future research. Additional descriptions of the AP software, tips on acquisition of requirements for AP, metrics and measurements in AP, and further elaboration of the application case studies, including descriptions of the experiment setups and guidelines on how to apply the AP methodology and software are appended at the end of the thesis. A glossary of terminology is also provided.

1.7 Publications

Parts of this research work have been published in various scientific journals and conference proceedings. Opiyo et al., (2002 c) presents a comprehensive state of the art review and clarifies the research problem. The theoretical fundamentals of abstract prototyping have been published in Opiyo et al., (1999 a); Opiyo et al., (1999 b); and
Opiyo et al., (1999 c). The AP application methodology has been covered in piecemeal in Opiyo et al., (2000 a); Opiyo et al., (2000 b); Opiyo et al., (2001 c); and Opiyo et al., (2002 a), and the results of the research on development of software tools to support AP activities articulated in Opiyo et al., (2000 c); Opiyo et al., (2000 d); Opiyo et al., (2001 a); Opiyo et al., (2001 b). The application case studies conducted to explore the validity of the AP concept have been published in Opiyo et al., (2002 b) and partly in most of the above-mentioned papers as proof-of-concept examples. Opiyo et al., (2002 d) partly summarizes the research work and results.
Quality Assurance of Design Support Software: A Literature Review

This chapter reviews the state of the art and practice in the development and quality assurance of the DSS products. First, the processes of development of the DSS products are studied and analyzed, and then the risk-reducing techniques employed during their development are reviewed. Based on the reviews and the analysis, a concept for pre-implementation testing of the DSS is proposed, and its key desirable characteristics are concisely presented.

2.1 Background

Designers began to use software tools for designing products in 1950's\(^1\) [Zeid, (1991); Horváth, (1998)]. Numerous software tools for supporting designers have been built ever since, most of them dedicated for use in the late phases of the design process, namely for drafting, modeling and engineering analysis. Providers are releasing versions after versions of various DSS packages in their efforts to improve services they offer. On the other hand, researchers continually develop new theories and computer-oriented methods, and numerous prototype DSS have been built in various research institutions. Like in the development of other software products, the developers of DSS use systematic development models and testing techniques, and adhere to practices such as usage of good programming styles and safer programming languages to assure quality. Other techniques in the areas such as requirements engineering and software prototyping as well as standards are also out there, and can be used in the development of DSS. Conceptual design, engineering analysis, detailed design, drafting and design documentation are typical design activities supported by the DSS [MacKrell and Herzorg, (2000); Flachsbart et al., (2000)].

This chapter presents and discusses strategies used in quality assurance of the DSS and the objectives are threefold, namely: (i) analysis of how the DSS evolves; (ii) reviewing the processes of development and testing of DSS; and (iii) concluding on a methodol-

\(^1\) This date is arguable. Some publications state different dates.
ogy for pre-implementation testing of DSS. The analysis is intended to reveal how the DSS evolves, as well as how their development process differs from the processes of development of other software products. Having known what it takes to develop DSS, various approaches used in search for quality are studied and the characteristic features for a new quality improvement methodology are eventually proposed.

This review is organized as follows. The following section analyzes the processes of development of DSS systems. It briefly describes how these systems evolve and it presents various strategies to ensure their acceptability. Section 2.3 presents a review of various concepts used to shape and direct software development activities, and investigates how far they cover aspects of the processes of development of DSS. A review of various verification and validation (V&V) strategies is given in Section 2.4. Section 2.5 reviews other measures, activities or norms that help assure quality. These include requirements engineering, involvement of various stakeholders in the development process, handling of knowledge, ensuring compliance to software quality standards, and measurement of software products. Finally, Section 2.6 summarizes findings of the review and Section 2.7 discusses and gives final words about the review, and presents the characteristic features of the projected concept.

2.2 Analysis of the Processes of Development of Design Support Software

DSS products are typically complex software products and they provide a large spectrum of capabilities and applications. To have a good understanding of the process of development and testing of these software products, various development and testing processes are studied and analyzed. Based on the reviews, generic features of these processes are eventually compiled. An overwhelming number of literature sources shows that DSS are based on engineering principles [Kreyman et al. (1999); Lee (1999); Zeid (1991)]. To develop DSS, the developers need theories, concepts or mental ideas about how the eventual software will work. Functions are usually bounded by principles and theories underlying a given field [Zeid, 1991]. Based on theories, various formal methods have been developed and different kinds of algorithms designed and implemented. For instance, for geometric modeling, a wide variety of supporting theories such as Coons patches, Bezier surfaces, Gordon surfaces, Overhauser surfaces, Bicubic patches, non-uniform rational B-splines (NURBS) and solid modeling [McMahon and Browne, (2000); Mortenson, (1997), Kasik, (2000)] loomed in different historical timelines and have been used as foundational theories in the development of various customized or general-purpose design support applications. Theory-intensive algorithms such as finite-element based and finite-difference based algorithms have been used in engineering analysis. Various concepts have been used in the areas of product data exchange and in geometry modifications. Publications and empirical evidences show that various development efforts are typically of unique cooperation be-
between the industry and the academia [Mortenson, 1997], but also individual developers or organizations often develop and refine theories and techniques.

Literature on how the DSS are developed by software companies is very sparse. Figure 2.1 shows a general phased empirical scheme, which is typically used intuitively in the development of DSS. What happens is that the developer starts by exploring possible solutions and then experiments with prototypes, selects the best alternative and improves it until signs of success show up, and afterwards implements an initial version of the software. Ad hoc strategies are used in accomplishing various activities within these phases. This approach is followed informally mainly in research-oriented as well as in commercial DSS development processes.

![Diagram](image)

**Figure 2.1: General scheme for development of DSS**

To give an insight on how the development progresses in the framework of this scheme, an approach followed in Dijk et al., (1998) is briefly analyzed. They were involved in the research and development of a computer-based tool for shape manipulation. Initially the problem was rather unclear, and some explorative work had to be carried out. They started by exploring possible solution concepts and came up with six candidate methods, which they evaluated through experimentation with prototypes. The development team knew some of the alternative solutions in advance and others were thought of, or found from literature. They used a test panel that consisted of practicing and aspirant product designers to evaluate the feasibility of the alternatives. One method was ultimately selected and adopted to be the underlying method for the shape modification functionality. The selected method was subsequently transformed into algorithms, and eventually codes for a pilot software prototype were written. These activities appear in many other DSS development projects (for instance, in the EREP project [Hoffmann, 1997], and there are many similarities across various DSS development processes.

Figure 2.2 depicts the process discussed above. DSS as well as other software used by scientists and engineers are usually the embodiments of mathematical models of physical phenomena [Kreyman and Parnas, 2002]. In the development of these kinds of software, knowing the needs and the desirable characteristics of the expected functionality and having translated them to requirements, the developers usually select or develop
foundational theories from scratch, and then formulate methods based on the selected or
developed theories. Typically, a large proportion of the development time is spent on
these activities. The methods are then transformed into algorithms, and finally codes for
pilot prototypes are written and the resulting implementation is tested. Formal V&V
methods used in software engineering are also used in these processes. There are also
several dedicated V&V strategies such as the experimental design projects method, the
research cycle procedure, and the quick evaluation method used in search for quality of
DSS [Vergeest et al., 2001]. In the work of Dijk et al. (1998), an approach that involved
prototyping solution concepts was pioneered and more systematically applied by Van
Elsas (1997), following the Research Cycle methodology [Vergeest, 1996]. The meth-
odology can be viewed as a pre-implementation V&V technique dedicated to ensuring
quality of critical functions of DSS.

![Diagram of DSS development process]

**Figure 2.2:** Detailed schema and feedback loops in the process of development of DSS

In conclusion, it can be said that:

- DSS are different in that they are based upon engineering principles or physical
  phenomena. The processes of realization of these kinds of software are typically *ad
  hoc* during conceptualization and design, and heavy in research. Interplay of the
  academia (who develop theories and methods) and the industry (who converts theo-
  ries into applications) is common. Often the problems are not clear and sometimes
  the developers themselves even do not know if they do not know something. Diffi-
  cult challenges are often thrown to the academia, often as contract research, or
  sometimes the research and development departments are summoned to take charge.

- The developers work with DSS in the design phase under various guises, which
  implicate diverse in-process abstract appearances of software in different contexts.
This is typical in research-oriented projects and is also common in business-oriented projects.

- Non-systematic and informal procedures are typically followed in reviewing early in-process implementations. However, it can be said that quality assurance strategies used in the early phase of development of DSS are mostly of usability nature, and the DSS developers generally pay scant attention to other desirable characteristics such as accuracy, robustness and integrity. In the later phases, and especially in the development of commercial software, the traditional techniques used in the development and V&V of software products are also used in the processes of development and testing of DSS.

The following sections review the risk-reducing techniques employed in conventional software development and study to what extent they embody characteristic features required in the processes of building DSS. It is generally understood that quality of products is determined during the entire development interval [Petschenik, 1985]. Two approaches, namely (i) assurance of the process by which software is developed, and (ii) verification and validation of various in-process implementations are followed to assure quality. These approaches as well as other risk reducing practices such as involvement of stakeholders and adherence to standards are reviewed and discussed.

2.3 Assuring the Process of Development of Software

Quality of software largely depends on quality of its engineering process itself [Beizer, 2001 a]. Hence, choosing, and consistently working within, a framework of a formal model or software management process can have a huge impact on the quality of DSS. This section presents and discusses various kinds of models, methods and process management strategies used in software development projects.

2.3.1 Software Development Models

Typical software projects follow concepts or mental models that shape and direct the development activities. Several process models for software development are available. These models present only the common stages of development, without prescribing, for instance, strict order or duration. Examples of these models include waterfall, program growth, component reuse, spiral, V, operational and rapid prototyping models [Lewis, (2000) and Jones (1990)]. Most software development models include requirements analysis and specification, design, implementation, testing and operation as the main phases [Melhart, 2000]. During the specification and analysis phase, goals are identified and requirements, which state what the software will do, are specified. In the design phase, sub-components of the system are identified, the goals are decomposed into targets and the final design, which depicts how the software will be implemented, is prepared. Afterwards, the sub-components are implemented according to the designs and units tested as they are completed. During the testing phase, classical tests such as performance tests, benchmarks and stress tests are conducted to determine whether
goals have been met. During the operation phase, the results of the application cases are used to fine-tune the solutions.

Examination of various publications in software engineering shows a consensus about the names of these phases. However, there is notable disagreement about the actual activities and products of each phase. Moreover, various models have different properties. For instance, some of the models are linear, others are recursive, some put emphasis on early phase activities, and others are for rapid application development. In the following sub-sections, features of various models are explored.

**Linear Models**

Linear models demand sequential determination of system's requirements, design and writing codes. Examples of linear models include the *waterfall* model [Royce, (1970); Jones, (1990); Boehm, (2000)] and the *V development life cycle* [Forbes and Thornton, 1999]. Linear models have held that software development proceeds in an orderly way through the phases of software lifecycle. Many software developers see linear models as useful and suitable for large software projects. However, there are also a number of limitations, which include, for instance, their weakness in maintaining user and other stakeholders' involvement throughout the development process, and postponement of the realization of tangible results and actual experience until late in the development process. Also, they put a lot of emphasis on documentation, they do not support reuse well, and they fail to support parallel activities satisfactorily.

**Incremental Models**

Incremental models assume that software development is a top-down or a bottom-up process in which parts of the eventual software are implemented one after another. Parts that give high impression such as the user interface or the database are implemented first, and the less impressive functions and lower level details come last. The unimplemented functions may be represented, for instance, by dummy choices on a menu. This allows for tests such as usability tests to be conducted early on. Another possibility is to integrate components of an existing software product to satisfy requirements of the anticipated software. In extreme cases, the process may involve assembling of the existing codes (that have previously been tested and used) without any modification, and using an interface or command language as glue. The *program growth* model [Brooks, (1987); Boehm, (2000)] and the *component reuse* model [Jones, 1990] are examples of concept models that demand software development to proceed in a paced, incremental fashion. Incremental models provide economical ways of creating new software. The challenge that the developers may face when using incremental models is how to interface components in an effective way and how to synchronize the needed modification to the existing code. One of the weaknesses of the incremental models is that bad design decisions can be institutionalized early on and it is difficult to change the underlying methods in the midst of a development process.
Recursive Models

Recursive models such as the spiral model [Boehm, (1988); Hendrix and Schnider, (2002)] and the evolutionary model [Lehman, (1980); Schneidewind (2000)] reduce software development to a cycle of recursive activities. The spiral model, for instance, demands recursive setting of goals and constraints, evaluation and resolving risks, development of the product, and planning for the next round. Typically the developers have to repeatedly: (a) investigate the problem and explore possible solutions, (b) evaluate solutions and identify risks, (c) resolve risks through experimentation with prototypes, (d) develop and test the product of the existing cycle, (e) plan for the next round of activity, and (f) review the results of the existing round of development. The development is in essence repetition of these activities to create increasingly complex versions of the final software product. The philosophy behind evolutionary development is that change or improvement is intrinsic in software and that software undergoes changes or improvements throughout its life (that is, improvements or changes to requirement specifications, design specifications, test plans and codes). The idea is that changes generated by the users and the environment, and the consequent needs for adapting software to the changes, are unpredictable and cannot be accommodated without iterations. In evolutionary development, no distinction is made, for instance, between maintenance and the initial development of software. The evolutionary model holds that software is a moving target and that large and complex systems are never completed, they just continue to evolve. The strength of the recursive models is that they reasonably minimize risks by identifying uncertainties repeatedly, most often by using prototypes of one kind or another to clarify the problems.

Early-Phase Oriented Models

The early-phase oriented models put emphasis on ensuring completeness and consistency of the evolving software, in particular the requirement specifications or designs. Specification or design requires extra efforts. Specifications can be transformed into designs or codes only when the developers feel they got them right. This can be accomplished interactively and the transformations can be effected automatically using software tools. The early-phase models provide quick results and allow early investigation of the envisaged software product. The operational model [Zave, 1984] is an example of the early phase oriented models. It concentrates on technical specifications from an algorithmic point of view rather than from an implementation viewpoint. The development passes through the analysis, specification and operation phases. The design, implementation and testing phases are ideally omitted. It starts right away with how software works. Codes are generated based on the specifications. This is rather orthogonal to standard norms of first describing what the software does and deferring definition of how as long as possible. Object oriented methods such as Object Oriented Analysis and Design (OOAD) and Universal Markup Language (UML) [Rumbaugh (2000); Wagner (2000)] also fall into the category of early-phase oriented models. Typically, they are intended to capture design decisions, and they operate at various levels of granularity. Freedom of the development process from unnecessary details and improved depend-
ability are the strengths of the early-phase oriented models. Furthermore, maintenance requires only modification of the specifications or the designs, with little testing. Some developers cite absence of motivation to plan as one of weaknesses of early-phase oriented models.

**Rapid Application Development Models**

It is typical and useful in some software projects to quickly have preliminary versions of software to help the developers gain insight into the nature of the problem and to experiment with alternative solutions. This kind of software realization is referred to as rapid application development. Software development concepts such as *rapid prototyping* [Luqi, (1989); McCracken, (1981)] involve development of quick-and-dirty versions of the anticipated software, often by reusing parts from previous implementations to create skeletal software. Little emphasis is given to quality characteristics such as efficiency or accuracy. The developers and the representatives of the prospective users evaluate preliminary versions of software with the understanding that they are not necessarily complete, final products.

**Summary**

Various kinds of models offer the developers alternative concepts for accomplishing software development activities. There is little consensus among the broad software engineering community concerning the nature and contents of the model most appropriate for software development [Moore, 2000]. Most models are considered to be inadequate on their own by the software process modeling community because the granularity of the process steps included is too large [Curtis et al., 1992]. The life cycle descriptions usually focus abstractly on the engineering of products and fail to specify the elementary building blocks necessary for managing and coordinating software projects. It is recognized that no matter which process model is chosen, it will not be a total solution to the software crisis [Brooks, (1987); Jones, (1990)].

Table 2.1 summarizes the main features of models widely used in the development of software products, including DSS. It can be deduced that:

- The current process models demand the developers to work with software in three different guises namely, specifications, design and codes [Figure 2.3].
- Most models guide the developers to first build executable prototypes and then test them. Such experimental software typically requires extensive efforts and resources to build.
Table 2.1: In-process implementations reviewed or tested in various phases

<table>
<thead>
<tr>
<th>Software development model</th>
<th>Phases of Development</th>
<th>In-process implementation reviewed or tested</th>
<th>Testing or review phase(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component reuse Evolutionary model</td>
<td>Specification, design, implementation, testing, operation.</td>
<td>Specifications, design, codes</td>
<td>Specification, design, testing</td>
</tr>
<tr>
<td>Operational model</td>
<td>Analysis, specification, operation</td>
<td>Specifications, design, codes</td>
<td>Specification, operation</td>
</tr>
<tr>
<td>Program growth</td>
<td>Paced specification, design, implementation and testing of parts of source code.</td>
<td>Specifications, design, testing</td>
<td></td>
</tr>
<tr>
<td>Rapid prototyping Spiral model</td>
<td>Specification, prototyping, testing</td>
<td>Specifications, codes</td>
<td>Specification, prototyping, testing</td>
</tr>
<tr>
<td>V model</td>
<td>Specification, design, implementation</td>
<td>Specifications, design, codes</td>
<td>Specification design, implementation</td>
</tr>
<tr>
<td>Waterfall</td>
<td>Requirements, analysis, design, construction, unit tests, validation, operation, evolution</td>
<td>Requirements, design, codes</td>
<td>Requirements, unit tests, validation</td>
</tr>
</tbody>
</table>

2.3.2 Automated Software Production

Costs involved in software development and testing are typically very high. Automation can have a massive potential to not only reduce costs, but also to increase quality and cut time-to-market. There are several automated (i.e. computer-based) software development and testing systems used in software engineering, variously known as Computer Aided Software Engineering, Software Engineering Environments, Integrated Project Support Environments; Meta - Computer Aided Software Engineering and Computer Aided Method Engineering [Gray et al., (2000); Zallar, (2000); and Robbins Redmiles, (2000)]. These systems encompass procedures, methods and tools which can be used to

![Figure 2.3: Feedback loops when using traditional software development models](Image)
develop, maintain and re-engineer software [Sharon, 2000]. The aim of these technolo-
gies is to achieve further automation of software production [Tate et al., 1992] just as
CAD/CAM systems have done to the engineering of mechanical and industrial prod-
ucts.

The specific aims are to improve software definition, design, production and mainte-
nance through usage and integration of software tools. The activities supported by the
automated development and testing systems are (i) front-end analysis and design (which
includes planning, requirements definition and designing); and (ii) development (which
include editing, compiling, debugging, and coding or application generation). Other
activities are (iii) management (which include configuration, project and processes
management), and (iv) supporting development (which includes re-engineering, main-
tenance and documentation). Typical capabilities provided by these systems include, for
instance, the ability to build a design or structured analysis model and an architectural
design, from which it can be traced how and where a requirement is fulfilled. In addi-
tion, models can be validated for consistency and logical integrity, and finally docu-
mentation of requirements can be produced automatically. The requirements tracing
utilities typically facilitate cross-referencing any type of information to the modules,
functions, data types, variables and constants in the model. The trace items can be the
system requirements or any other information. Cross-referencing can therefore define
functions that implement certain requirements or conversely, the requirements fulfilled
by functions.

The introduction of automated software production is seen by many organizations as
providing a means of improving consistency, repeatability and definition of their soft-
ware process, as well as quality and productivity. However, automation has its limita-
tions. For instance, software design is not simply a process that can be automated by
transforming specifications. It involves complex decision making tasks that require ex-
perienced and skilled developers. Also, for a requirement analysis system that uses the
principle of cross-referencing to relate the requirements to functions, one of the prob-
lems is that not all requirements relate directly to the functions. The capability of such
systems to trace whether requirements that are not connected to functions have been
fulfilled is therefore limited. It is generally understood that the existing automated soft-
ware production methods in most cases are not sufficient on their own [Tate et al.,
1992]. However, when integrated with suitable metrics and a software process model,
they can form an effective test bed for modeling, measurement and management of soft-
ware projects.

2.3.3 Software Process Management Concepts

Some software development projects are typically complex and large, and formal meth-
ods for managing activities are necessary. Several formal processes and meta-processes
are available and can be adopted. Examples of technologies for software process man-
agement (SPM) include, for instance, *Rational Unified Process*, *Capability Maturity
Model (CMM)*, *Personal Software Process*, *Team Software Process*, and *Tailoring a
Measurement Environment* paradigm.
The SPM strategies bring into software development organizations techniques for assigning and managing tasks and responsibilities. They enhance team productivity and deliver best software development practices. Some of them, for instance, Rational Unified Process comprise tool sets with which, for example, the developers can visualize execution at the design level. Such a tool set offers a blend of testing and inspection capabilities in the sense that the scenarios represented by test inputs drive execution of a design model. CMM [Paulk, et al. (1993); Dymond, (1995); Jalote, (1999); Tate et al., (1992); Melhart (2000)] describes the elements of an effective software process and provides a framework for assessing capability of an organization to develop software. Its goal is to improve the ability of software organizations to meet goals for cost, schedule, functionality and product quality. Although intended for organizational assessment of key process areas, CMM levels can be adopted to inspection processes [Rodgers and Dean, 1999]. Personal Software Process is a method for individual developer improvement, while the TSP, which requires all team members to be Personal Software Process proficient, specifies roles and approaches to development [Ward et al., 2001]. The Tailoring a Measurement Environment paradigm [Basili and Rombach, (1988); Tate et al. (1992)] provides stepwise procedure for improving software processes by tailoring them to goals and environments. While these process models seem to be promising, they are generally considered to be most appropriate for large organizations with substantial resources.

SPM techniques provide customizable frameworks, which can be adapted to various ways companies work. Customers and users are represented in various development activities (such as in requirements workshops and in reviews). Some of the SPM techniques contain roadmaps, which walk the developers through how to use them in software projects. Such roadmaps provide the developers with instant access to specific guidelines, for example, for user interface design, architectures and patterns, programming, and performance testing. One of the limitations of the SPM methods is that they provide only higher-level guidelines. The lower level guidelines are not defined and the individuals are supposed to decide about details of the lower level activities. Each organization must adapt the technique in question to its own processes. It is also recognized [e.g., in Ward et al., 2001] that different software process models operate at different levels with different degrees of formality, and that common sense dictates that processes should be established and fine-tuned to match the needs of the organization.

2.4 Software Verification and Validation

Software developers need to check and to know how well the incidental products such as specifications and designs or the final software have been implemented. This can be achieved through verification and validation, which among other things involve conducting (static and/or dynamic) tests as well as experimentation with prototypes. The process of checking the consistency and completeness is known as verification while the process of assessing the likely extent to which the anticipated software will satisfy
user requirements is referred to as validation. Jagdev et al. (1995) says that verification ensures accurate translation of the conceptual solution into an operational computer program while validation provides confidence in the verified implementation. Thus, validation is a global term that embodies other terms (such as testing and verification) as specific aspects. Both verification and validation (V&V) typically involve assessment of the codes, designs or models of the envisaged software to find out whether it will meet the requirements specifications. This covers individual units, integrated units and the entire software system, and it is typically performed both by the software developer and the end user [Jagdev et al., 1995].

2.4.1 Software Testing

Software testing is part of the larger discipline of software engineering and it is defined as a planned risk reducing activity that takes place during software development, operation or maintenance [DeMilllo, 2000]. Such activities include, for instance, requirements analysis, experimentation with prototypes, static reviews of designs or other intermediate implementations, static and dynamic analysis of software, and re-testing during maintenance. Testing is an integral part of the validation process [Jagdev et al., 1995] and it is generally recognized as one of the key activities in software development. This activity has a strong influence on quality [Eickelmann and Richardson, 1996]. Almost all frameworks for the development of software products include testing as one of the life-cycle phases [Table 2.1]. The primary objectives of tests are (i) to ensure that the eventual software ultimately meets the needs and expectations of the customer, and (ii) to check the consistency and completeness of the implementations against technical requirements or constraints. Tests help the developers (i) to discover bugs at various levels of system complexity; (ii) to learn how future users foresee new concepts; and (iii) to find out in advance how the implemented system behaves in various practical situations. The result of tests is a verdict on whether or not the implementation in question fulfils the requirements of the users as well as the constraints stemming from technical infrastructure elements (such as databases, protocols etc.).

Software testing can be static or dynamic. In static tests, the analysis is done without running the program while in dynamic tests, codes are executed with the representatives of input, and the output is compared with the expected results. Testing typically consumes large amounts of resources, in the magnitude of 50% or more of the development budgets [Fairley, (1985); Shepard et al., (2001)]. Various kinds of testing techniques are available and the area is being actively researched. We categorize the existing testing techniques into two groups, namely classical testing techniques and review methods.

2.4.1.1 Classical Tests

Risk reducing activities performed to assess completeness, consistency or usability of specifications, designs or codes are referred to in this review as classical testing. These include (a) unit tests (testing at the lowest level to discover bugs in codes); (b) integration tests (testing interfaces of units to assure compatibility); (c) system tests (testing
the entire system to check for system bugs); (d) testing to requirements (testing from
the users' perspective to verify operability of features); (e) stress testing (subjecting a
system to unreasonable load to see how it performs); and (f) reliability tests (tests car-
ried out under a user profile to determine how a software product can be used). Others
are: (g) regression tests (carried out on a new version to determine whether apart from
changed portions, it performs as the previous version); (h) performance tests (testing
processing delay); (i) usability tests (checking the adequacy of human-machine inter-
faces such as icon style, menus and so forth); and (j) operational tests (performed be-
fore official release to find bugs not discovered earlier) [Agarwal et al., (2000); Beizer,
(1999); Bolton, (1999); DeMillo, (2000); Fairley, (1985); Forbes and Thornton, (1999);
Gabbard et al., (1999); Jones, (1990); Musa, (1998); Virzi et. al. (1996); Whittaker and
Voas, (2000)]. Most tests are typically performed after implementation and some of
them can be conducted in earlier phases as well.

As shown in Table 2.2, most classical tests typically require codes to be written. This
demands significant investments in terms of development time and money. A great deal
of resources can be saved if tests are conducted prior to coding. Another issue is that
some of the testing methods are coarsely described. Their sub-activities remain ad hoc,
and as a result, the success of such tests depends on skills, experience and knowledge
of the developers. Such informal methods can be unreliable and limited, as it is very
difficult to sufficiently take into account all attributes such as those associated with
functioning as well as other quality characteristics of the software.

<table>
<thead>
<tr>
<th>Testing technique</th>
<th>Products tested or reviewed</th>
<th>Review or test subjects</th>
<th>Testing or review phases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration tests</td>
<td>Codes</td>
<td>Developers</td>
<td>Implementation</td>
</tr>
<tr>
<td>Operational tests</td>
<td></td>
<td>Developers, users</td>
<td>Operation</td>
</tr>
<tr>
<td>Performance tests</td>
<td></td>
<td>Developers</td>
<td>Testing</td>
</tr>
<tr>
<td>Regression tests</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reliability tests</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stress tests</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System tests</td>
<td></td>
<td></td>
<td>Implementation</td>
</tr>
<tr>
<td>Testing to requirements</td>
<td>Specifications, codes</td>
<td>Developers, users</td>
<td>Design, testing</td>
</tr>
<tr>
<td>Unit tests</td>
<td>Codes</td>
<td>Developers</td>
<td>Implementation</td>
</tr>
<tr>
<td>Usability tests</td>
<td>Specifications, designs, codes</td>
<td>Developers, users</td>
<td>Design, testing</td>
</tr>
</tbody>
</table>

In conclusion, it can be said that:

- Most testing techniques require some sort of coding. This makes them lengthy and
  expensive processes, in particular, if a suitable implementation is not achieved in
  the first trial.
• For some testing techniques (for example, usability tests), only the main activities are described. The elementary activities are not formally defined, and this makes them somewhat subjective.

2.4.1.2 Review Methods

A review is a procedure for collection and analysis of information about quality of a solution concept or the final software product. Reviews (also known as inspections) are conducted to determine whether the proposed or the implemented functionality meets customers as well as technological demands. The concrete objectives of reviews are to: (i) determine the potential effectiveness of new methods and tools, (ii) study the operation of new products and the users' attitude towards them, and (iii) find out suggestions for improvement. Uden (1995) categorized review techniques into five groups as follows: (i) analytical, (ii) survey, (iii) expert evaluation, (iv) observational, and (v) experimental evaluation. There are also other types of reviews that provide empirical information about specific quality characteristics. These include, for example: (a) scenario tests (for studying usability); (b) systematic tests (testing the functionality, that is performance of the system and program features); and (c) feature tests (comparing systems of the same type).

Each of the above-mentioned reviews may be useful if applied appropriately, and sometimes combinations of different techniques can be more effective. In fact, there are no hard and fast rules in this, as each method has its particular strengths and weaknesses. Factors that can be considered in distinguishing review methods, and therefore help making an appropriate choice include (1) the stage in the life-cycle in which the review is carried out and (2) the style of review. Others are (3) the levels of subjectivity or objectivity of the method, (4) the type of measures provided, (5) the information provided, (6) the immediacy of responses, and (7) the resources required. However, it is not always immediately clear which method is the most appropriate or which parameters must be considered in order to have an effective review.

2.4.2 Controlling the Levels and Amounts of Details

Typically, in most cases, software projects are complex undertakings. Techniques for handling the inherent complexity during development or testing processes can therefore be helpful. Abstraction and prototyping are the two concepts commonly used to manage and cope with complexity during the engineering of software. They center on controlling the levels and amounts of details.

2.4.2.1 Abstraction

There are numerous definitions of the terms 'abstract' and 'abstraction'. For example, according to the Longman Dictionary of Contemporary English (1984), abstract means 'a shortened form of something, or something that does not try to represent an object as it would be seen' and abstraction is 'the act or action of abstracting'. From engineering viewpoint, abstraction is an analysis mechanism in which a complex reality can be evaluated based on a simplified model [Yourdon, 1994]. Abstraction as a process de-
notes extraction of essential details about an item or group of items, while ignoring inessential details [Berard, (1993); Pahl and Beitz, (1996)] and allows people to deal with concepts apart from particular instances of those concepts [Fairley, 1985]. Abstraction represents essential characteristics of an object that distinguish it from all other objects and thus provide crisply defined conceptual boundaries relative to the perspective of the viewer [Booch, 1991]. There are several forms of abstraction. For instance, functional abstraction generalizes a certain sequence of instructions, data abstraction treats certain named collection of data, and control abstraction implies control without specifying internal details. The notion of abstraction supports both creativity and systematic thinking [Pahl and Beitz, 1996] in engineering design, and can as well be quite useful in the process of reviewing early implementations of DSS.

2.4.2.2 Software Prototyping

Prototyping can be described as an experimentation process that involves usage of prototypes as means to facilitate discussions or clarification of concepts. A prototype typically has the look and the feel of what the software product will be like and little or none of the functionality [Melhart, 2000]. Prototypes reflect chosen aspects of software [Berzins, 2000] and provide a simplified view on the eventual system or its key parts. Budde, et al. (1992) described a prototype as an operational model that implements certain aspects of the imagined system. The common purpose of prototyping is to reduce uncertainty about properties of the anticipated software. It is recognized that the main incentive of using prototypes is economic [Berzins, 2000]. Scaled-down models provide a less expensive, yet concrete basis for discussing difficulties, clarifying problems or preparing discussions between the developers and the users or the management. In software development projects, it is typically not expected to get a fully functioning system the first time out. What is expected in early runs is prototypes, which bring the knowledge needed for building the system to the developers. Prototyping is traditionally applied to provide feedback to the developers [Budde et al., 1992] and is carried out particularly when the developers don’t know in advance what kind of knowledge might be needed or where requirements are initially unclear [Howard, 2001]. Software prototyping techniques are used in detailing requirements, in feasibility studies and in testing.

There are several prototyping methods in software engineering. These include rapid prototyping, reusable (evolutionary) prototyping, modular (incremental) prototyping, horizontal prototyping, vertical prototyping, low-fidelity prototyping and high-fidelity prototyping. Rapid prototyping involves quick implementation of new designs, which are then evaluated and thrown away, while in evolutionary prototyping, the effort put into constructing the prototype is not wasted, since parts or even the whole prototype is used to make the actual software product. In modular prototyping, few parts of the system are implemented first to serve as prototype and new parts are added as development progresses. In horizontal prototyping, prototypes cover a large breadth of dummy features and functions, while in vertical prototyping, prototypes cover only a narrow slice
of features and functions that normally do not work. The process of building physical prototypes on paper and testing them with real users is called low-fidelity (lo-fi) prototyping, while high-fidelity (hi-fi) prototyping involves building prototypes using such tools as demo-builders, multimedia tools or high-level programming languages [Virzi et al., (1996); Rettig; (1992); Fay, et al. (1990)]. The advantage of lo-fi prototyping is that it effectively helps educate developers to have a concern for usability and formative evaluation. It also reduces the number of times the developers get to refine their designs before committing to code. Furthermore, lo-fi prototypes are relatively cheaper and do not take too long to build or change. The hi-fi prototypes are used for such purposes as selling ideas about the expected software, testing 'the look and the feel', proofing concepts, and testing changes to an existing system. They take too long to build or change, and when used in testing, subjects tend to comment on the 'fit and finish' issues rather than on the underlying concepts.

In relation to traditional software development models, prototyping is typically part of requirements specification engineering and design activities [Melhart, 2000]. It allows the developers to test quality and functional attributes of the anticipated software at the specification and design phases. This is also the case in the development of DSS, where prototypes are realized mostly to verify specifications and design concepts. However, there is vivid evidence in the literature that in practice proposals for new methods are typically evaluated using pilot software implementations [Dijk et al., 1998]. Experimentation using partially or fully implemented prototypes tends to be lengthy and expensive.

2.5 Other Software Quality Assurance Issues and Strategies

Experiences show that in order to realize software with an acceptable level of quality, certain other precautions or steps must be taken into the consideration. For example, requirements need to be specified well early on, customers must be involved in the development process and the developers themselves must be knowledgeable about the problem and the expectations. Furthermore, standards provide procedural yardsticks for development and evaluation of software products.

2.5.1 Requirements Engineering

Requirements can be defined as descriptions of what software will do, and may consider the anticipated software or may include models of the larger system of which the software is to be a part. Requirements engineering is part of software engineering discipline, which is concerned with elicitation of goals to be achieved by the envisioned system, operationalization of such goals into specifications of services and constraints, and assignment of responsibilities for the resulting requirements to agents such as humans, devices, and software. It is now recognized as the key process in establishing a concept of the envisaged system and help managing convergence of stakeholders interests, achieving shared understanding of issues involved in realization of the system, and documenting this understanding [Jarke, 1998]. Specification languages such as Larch, Z
and Vienna Development Method (VDM) are used in various specification problems. Other techniques include those for system modeling, from semi-formal to formal. These techniques have relative strengths and weaknesses in requirements specification, notably, their limited scope and lack of methodological guidance. Recent efforts to overcome this kind of problems through specific techniques such as goal-oriented elaboration of requirements, multi-paradigm specification, handling of non-functional requirements, management of conflicting requirements, and handling of abnormal agent behaviors have been reported. Requirements analysis helps crystallize the customers' possibly vague ideas and reveals contradictions, ambiguities, and incompleteness in the requirements [Wing, 2000 a].

Publications report on four ways of formulation of requirements. These include: (i) trial and error empirical methods (that is, manual formulation of requirements from scratch based on the developers' experience and knowledge around them); and (ii) probabilistic guesses. Others are (iii) analogical methods (that is analogies between a given task and past case solutions); and (iv) algorithmic and artificial intelligence techniques. It is recognized that requirement specifications should be logically satisfiable and practically realizable, and should capture the full and precise intent of the specifier [Wing, 2000 a]. Getting high-quality requirements is difficult and critical. Detailed descriptions of the characteristics that good requirements should possess are available in many publications [for example in Moore, (2000)].

Requirement specifications have enormous influence on quality of software products. The stakeholders' requirements as well as technical constraints are used as criteria for testing designs and implementations. Software products can be regarded as good if requirements (quality, functional etc.) are fulfilled to a satisfactory level. Quality attributes implied by requirements are parts of standards for measurement in software projects. According to Glass (1992), software quality is measured via a set of attributes that are characteristic of high-quality software and the developers are responsible for building the desired attributes into the requirements model. Typical characteristics commonly taken into consideration are: completeness, correctness, dependability, efficiency, maintainability, portability, robustness, testability and usability. Some characteristics can be in conflict with others, and often trade-offs among desired quality attributes need to be made. The choice of characteristics to require in the final software product must be based on the value they add for the customer.

Despite the recent advances in the requirements engineering field, in practice, most software development projects typically do not possess an adequate requirements specification [Moore, (2000); Standish Group's Chaos Report, (1994)]. Many publications argue that the majority of encountered bugs, including the most serious bugs, lie in poor quality of problem specifications. To overwhelming degree, requirement specifications are generally unclear, incomplete, inconsistent, lacking in detail and riddled with ambiguities. Lack of user input and incomplete and/or changing specifications are pointed out as the main reasons for failure of most unsuccessful projects. If the stakeholders do not have adequate input then incomplete needs analysis, bad requirements and bad
specifications are inevitable. This often leads to (i) cancellation of projects before completion, (ii) delivery not according to expectations, and (iii) a tendency of development processes to become excessively lengthy. There is a general understanding in software engineering that finding an error and fixing it in the requirements specification phase is much cheaper than doing this in the maintenance phase. This explains why it is important to formulate consistent and complete requirements early on.

2.5.2 User Centered Quality Assurance

Involvement of representatives of the users of software products in the development process is recognized as a valuable practice. It significantly helps user acceptance of the final software product. The Participatory design and contextual inquiry concepts bring user aspects in software development. Participatory design is a set of diverse ways of thinking, planning, and acting through which people make their work, technologies, and companies more responsive to human needs. The idea behind the participatory design methodology is that successful application of software strongly depends on how the users will use it. As a methodology, participatory design [Namioka and Duog, 1993] addresses user’s involvement aspects in software development by making them part of the development team. Participatory design focuses on envisionment and evaluation of activities [Bayer and Holtzblatt, 1998]. The objectives of participatory design include mutual learning for the user and the developer, application of design tools familiar to users, envisionment of future work situations, and grounding the analysis to the practice of the user [Greenbaum and Kyng, 1991]. Effective participatory design requires common environment-shared media of analysis and design as well as shared terminology with which the user and the developer can effectively communicate [Chin et al., 1997]. It differs from other user-centered procedures in that the users and the developers co-determine the appropriateness of the solution proposals. Uden (1995) describe two possibilities of implementing participatory design principles, namely ‘mutual reciprocity’ and ‘design by doing’. In the mutual reciprocity approach, the users and the developers teach each other about the prospective work practices and technical possibilities through a joint experience, while in design by doing, the focus is on interactive implementation, modeling and testing to provide hands-on experience to future users. In most cases, both of these approaches are pursued largely with low fidelity prototypes, which users are familiar with and which they can learn and employ themselves.

Contextual inquiry [Bayer and Holtzblatt (1998); Coble et al., (1997); Holtzblatt and Bayer, (1998); Holtzblatt and Jones, (1993)] is a field technique that is used to explore usability problems of software tools experienced by users in their working environment. It enables the developers to collect detailed customer data needed for the project. The developers focus on asking questions while subjects perform the tasks. The principal method used is contextual interviews, in which the subjects and the developers discuss the user’s goals, ways of working and problems encountered when using the envisaged system. The developers and the customers collectively analyze information (presented in the form of text notes, audio clips or video pictures) gathered when the subject uses prototype of the envisaged system. The problem with contextual techniques is that the
basic underlying procedures are informal. Also, often it is not immediately clear, for instance, in what development phase contextual inquiry should be conducted, and how subjects should be chosen.

2.5.3 Knowledge of the Developers

In its broadest meaning, the term knowledge is used to refer to any organized body of purposefully aggregated information that can support nontrivial reasoning and problem solving [Hayes, 2000]. This includes, for example, repositories of information, in various forms of representations, intended for human use for various purposes. Such knowledge can be deposited in various places, for example in DNA, brains, hardware, books, and in software [Armour, (2001 b); Dempsey et al., (2002)]. A gear-design engineer, for instance, could keep his/her knowledge in his/her brain, could write a book or could write codes for software that does tasks s/he usually does. To those people who have no background in gear design, a software version is definitely the most valuable alternative, as in the ideal case they can do what the gear designer does without knowing what the gear designer knows. Software is more flexible and in certain cases can be used outside its intended domain.

In order to reuse knowledge, it must be represented conveniently. The term 'knowledge representation' is widely used in the field of artificial intelligence, and fundamentally it means a surrogate, a substitute for the thing itself, used to enable an entity to determine consequences by thinking rather than acting, i.e., by reasoning rather than taking action in it [Davis et al., 1993]. Furthermore, it can be described as a set of ontological commitments and it is a fragmentary theory of intelligent reasoning, expressed in terms of three components, namely: (i) the representation's fundamental conception of intelligent reasoning; (ii) the set of inferences the representation sanctions; and (iii) the set of inferences it recommends. It is a medium for pragmatic and efficient computation, and one contribution to this pragmatic efficiency is supplied by the guidance a representation provides for organizing information so as to facilitate making the recommended inferences. Knowledge representation is also a medium of human expression, i.e., a language in which things are said. In the context of this work, the term 'knowledge representation' is used somehow differently from the artificial intelligence context in that it neither embraces ontological commitments nor includes intelligent reasoning [see Chapters 3 and 4].

Some practitioners as well as researchers in software engineering associate software development processes with acquisition of knowledge. For instance, Armour (2000, 2001 a, 2001 b, 2001 c, 2001 d) says that software development is a knowledge-acquiring activity and that the hard part of building software is in acquisition of the necessary knowledge. The more knowledgeable the development team, the greater the chance of creating quality software. In software development projects, the developers have different degrees of knowledge about the problems, expectations and solutions. There are projects in which the developers already have solutions while on the other extreme end; there are projects that do not even have a problem statement leave alone
the solutions. The former projects can have a highly explicit and well-controlled process to implement the solutions while in the latter ones the developers cannot use a well-defined process because it is not even known which process might work.

Software development processes are usually knowledge-intensive. However, sometimes it is difficult to retrieve or find knowledge beyond the developers' reach. Techniques available in the field of information retrieval for structuring, analysis, organization, storage, searching, and dissemination of information can be employed to make information available to the developers. Clearly issues such as chunking, representation and structuring of knowledge have to be resolved. In retrieval operations, a search request is typically constructed by choosing appropriate keywords and content terms and appropriately interconnecting them using Boolean operations (that is, 'and'; 'not'; and 'or') to express the request of the user [Salton, 2000].

Research in commercial organizations and universities has recently enabled development of systems that allow users to express their search in natural language, without the need for specific syntax. Also, the retrieval techniques have been extended and used in new search applications such as for the Web. Furthermore, advances in information technology have lead to evolution of techniques for supporting working methods such as collaborative development. Examples of these techniques include TeamRoom, IBM's Lotus Notes, Microsoft's NetMeeting, and TeamWave [Fuchs, et al. (2002); Tomek and Giles, (1999)]. Typical features of these tools include, for instance, a user interface with collaboration options; real-time conversations via text; a shared whiteboard; shared files, URL links, and images; point-to-point audio and video links; e-mail facilities that enable users to send notes; and status tracking of documents. Several means of handling large amounts of general reusable knowledge in fields such as product design and manufacturing are available. One of the main limitations for most of the knowledge acquisition and GroupWare systems is that they focus primarily on capturing and supporting knowledge structuring and exchange rather than on reasoning with knowledge [Ramesh and Bui, 1999].

2.5.4 Software Quality Standards

A standard can be defined as a formal normative specification that is established based upon the consolidated results of science, technology and experience by consensus and approved by a recognized body, which provides for common and repeated use, rules, guidelines or characteristics for activities or their results [Cargill, 2000]. One of the purposes of quality control in software engineering is to ensure compliance with software standards. Standards arise either from certified standard activities (de jure standards) or by force of practice (de facto standards and publicly available specifications - PASs) [Cargill, (2000); Rada and Craparo, (2000)]. Standards establish operating procedures for providing quality services. The advantage of using standards to describe software quality is that they systemize and guide the development process and enable software to be developed in a way that leaves no room for ambiguity. The possibility of misunderstandings when using standard norms is generally minimal. Standards provide procedures that are collectively grouped into a model for quality assurance in design,
implementation, testing, marketing, installation and servicing. The standard operating procedures can be customized to the individual needs by simply adding or deleting sections, as appropriate. There are hosts of software quality standards around. When a company that develops software looks for a standard, the issue is not that one standard is better than the others, but rather that a software development company chooses a particular standard and consistently works with it. If adopted, standards improve quality of software products and at the same time create a large market instead of many fragmented markets.

Official Standards

The International Standardization Organization (ISO) and the Institute of Electrical and Electronics Engineers (IEEE) Inc. are actively involved in the development of *(de jure)* standards used by software developers around the world. ISO software engineering standards includes, for instance, the ISO/IEC 9126-1 quality model (developed jointly with the International Electrotechnical Commission - IEC), which pertains to evaluation. It is concerned primarily with definition of quality characteristics to be used in the evaluation of software products. It sets out six quality characteristics, namely, functionality, reliability, usability, efficiency, maintainability and portability. These characteristics are not rigid or unchangeable, and they can further be broken down into sub-characteristics. They serve as a checklist, guiding the evaluator in his/her attempt to decide and define what characteristics contribute to quality and therefore should be measured when carrying out an evaluation. The problem is that each quality characteristic is very broad and it is difficult to know when a quality characteristic is fully covered. Other *de jure* ISO standards include (i) ISO 12207 that describes the major components of software life-cycle, their interfaces with one another and the relations governing their interactions, and (ii) ISO 9000 that describe quality of products.

The IEEE *de jure* software standards for streamlining software development processes include a standard for software quality assurance plans (730-1998); a standard for software configuration management plans (828-1998); and a recommended practice for software requirements specifications (830-1998). Others are: a standard for software unit testing [1008-1987 (1993)]; a standard for software verification and validation (1012-1998); guides to classification of software anomalies (1044.1-1995); a structured format for software development plan (1058.); and many others. Other numerous international and local organizations and governments around the world write various software engineering standards or help organizations deal with existing standards.

Standards Arising by Force of Practice

Standards arising by force of practice (that is, *de facto* standards) are those specifications that have gained market acceptance and are recognized as yardsticks for software realization. The so-called *publicly available specifications* (PASs) are unofficial specifications, which the market has accepted, but have not yet been through the formal standardization process. Continued survival of a *de facto* standard usually depends on their professional significance as well as the amount of marketing. Microsoft Solution
Framework (MSF) is an example of de facto standard that has been chosen by many software development companies. PASs are normally created by consortia\textsuperscript{2} and they operate in areas of fast moving or contentious technologies where standard organizations have been judged to be too slow.

The de jure or de facto standards specifically dedicated to DSS are very sparse. The available DSS standards include, for instance, those used in product data exchange between CAD/CAM systems such as STEP and IGES [Bloor and Owen, (1995); Owen, (1993)], which are relevant to the DSS, but not for guiding their development processes. When an organization that develops DSS looks for a standard, they are obliged to choose from among traditional standards used in software engineering.

It can be said that DSS developers can benefit enormously from the application of established standards and procedures to assess compliance with specified objectives and to avoid risks of undesired behaviors. It must, however be noted that using standards is not a panacea to DSS quality assurance problems. It is recognized that most often standards are not definitive answers to all problems, for instance, they fall short in addressing and detecting potential design conflicts [Mercuri, 2002].

2.5.5 Metrics, Measurements and Consensus Making

Software metrics are units of measurements of software. They form the basis for models of software development process and make it possible for software to be compared and analyzed quantitatively. Without metrics, it would be difficult to understand and improve the complex process of software development and meet goals [van Verth, 2000]. In practice, software metrics are among other things used to evaluate acceptability and quality of software, and to determine whether specifications have been met and users demands satisfied. They are also used to provide feedback during the development process and to signal when quality standards or software practices have been violated and when software needs to be corrected.

Some practitioners have, however, received software metrics with a great deal of skepticism. This is largely because of poor methodology and faulty data that they are often based upon. For example, many experimental studies have been criticized for using faulty designs and improper statistical techniques [Beizer, 2001 a]. Part of the problem is that there is no concrete theoretical basis upon which to develop metrics. Many models and metrics are proposed by the individuals for specific situations and tested in a limited context. They cannot be readily generalized to other situations and contexts. In general, metrics do not scale or reliably transfer from one project to another.

Consensus making is a formal or informal activity that recurs in most phases of the software development process. For example, in the early phases, usually concepts are informally reviewed and the appropriate ones are selected while at the design phase,\textsuperscript{2}

\textsuperscript{2} Consortia are groups of companies formed to unify industries around a singular technical specification [Cargill, 2000]. A consortium tends to focus on the creation of a unified specification that corresponds to specific industrial problems, for example, language standardization.
alternative designs may be available and the developers have to choose suitable one. Metrics plays an important role in consensus making. Several consensus-making systems have been proposed. Some of them are somewhat specific [for example, those proposed in Kato and Kunifuki (1997) and in Kawakita (1975), which are for use in requirement acquisition and analysis], while others [for example, decision matrix methodology] can be used across the entire development process. There is little agreement concerning which consensus making systems are most suitable. However, it is generally agreed that the decision matrix approach [Steward, 1981] can provide reliable results, provided that right criteria are used and the appropriate subjects are involved.

2.6 Summary

It has been shown in this review that one of the distinct characteristics of the processes of development of DSS is that their design phases are broad and early activities are highly ad hoc and research oriented. They typically involve selection or creation of theories, formulation of methods, designing of algorithms, and writing codes for pilot prototypes. Most often the problems are unclear and the solution concepts are unknown. As for development and testing of DSS, the traditional software development models and testing strategies are typically used. This is done regardless of fundamental differences between DSS and other software products. It is echoed in many articles [for example in DeMillo, (2000); Jones, (1990)] that in traditional software development models, the developers work with software, which appears first as specifications, then as a design and finally in form of codes. Almost all software development concepts or mental models demand the developers to work in this framework and even tests and reviews are conducted to exclusively ensure consistency and completeness of these in-process products. It has been shown that the actual software product, specifications of its design, or models of its intended use are typical articles that are dealt with in conventional tests or reviews. It is understood in the software engineering community that large-grained life-cycle descriptions used in software development do not necessarily correspond to the processes actually performed during software development [Curtis et al., 1992]. They typically represent high-level plans, and do not contain information that systematically explains how to proceed, for example, when working within the phases of software process models. It can therefore be said that the DSS developers' community require finely defined models that can cope with the needs.

Also in this review, various systematic software process management strategies and testing techniques have been presented and discussed. These, along with good programming styles, safer programming languages, standards and many other dedicated measures have been created to ensure quality of various software products. It is also universally recognized that the key aspect to insuring quality is interaction with the customer. Literature [for example Doll and Deng, 1999] acknowledges that knowledge utilization is a social activity and that complex problems cannot be solved through spe-
cialists thinking and working in isolation. User participation is widely accepted as essential to developing successful applications.

Despite all of the advances in software engineering and all good software practices in place, project failure still remains a critical challenge to the software development community [Keil and Robey, 2001]. Significant numbers of projects are neither delivered on time, nor on budget, nor with the promised functionality. In some instances the developers are still delivering buggy and unreliable software systems, and worst still some users believe it is normal for software to be buggy [Booch, 2001]. As for using the available techniques and measures in quality assurance of DSS, it can be said that they cannot scale to robustly support this endeavor. Most of them are not sufficiently described to precisely address the needs in the processes of development of DSS. After all, some of these techniques have been developed for specific problems and purposes, in a much simpler era [Amour, 2001 d]. Amongst techniques that have been reviewed, two of them seem to comprise concepts that can be extended and used in pre-implementation testing of DSS (that is, in quality assurance of the in-process implementations of DSS at the design phase). These are (a) user-centered techniques, and (b) prototyping. Adapting these techniques to the DSS development creates effective ways of ensuring quality. When prototyping is applied, it should focus on experimentation with the in-process implementations, and should involve both the users and the developers.

In conclusion, quality is defined as the degree of excellence of something [Glas, 1992]. This implies a subjective factor, namely, any software project can be found lacking if measured against a vague notion of what quality is. Various testing strategies can be used to determine whether software satisfies its specification, and it is recognized [for instance in London and Craigen, 2000] that each technique provides a different level of assurance. The software engineering community in general concedes that there is no silver bullet development model or V&V technique that will work for all software organizations and for all software projects [Amour, (2001 b); Brooks, (1987); Brooks, (1995); Howard (2001)]. And it is universally understood nowadays that different problems require different solution approaches, and methodologies must be tailored to the problem domains [refer, e.g. to Glass, (2002); Vassey and Glass, (1998)]. It is also generally understood that bugs cannot completely be eliminated [Lieberman and Fry, 2001]. In a typical software engineering process, the design phase in particular accounts for significant additional faults [Eick et al., (1992); Hevner, (1996); Humphery, (1989)], and should therefore be one of the focal areas of research in quality assurance of DSS.

2.7 Discussion and Concluding on a Concept for Pre-Implementation Testing of Design Support Software

In spite of their importance, the DSS have not received attention in the software engineering community, in terms of research work on assuring their quality and trustworthi-
ness. Subtle, but important, flawed DSS can be put into service. It is often wrongly assumed that designers who use DSS can countercheck the results generated by these software products. Counterchecking the outputs of DSS can, however, be a massive undertaking for the designers, especially for complex product design problems. Normally what they can afford is just few rough checks, and there is a big chance that faults will remain undetected. In addition, designers have the habit or tendencies of trusting results generated by computers, and this can lead to serious adverse effects. It is important to note that designers who use DSS might not notice that software is not a trustworthy implementation of the engineering principles they wished to use. Systematic methods for development as well as verification and validation of DSS in advance would therefore be very useful.

The processes of development and testing of DSS have been studied (and presented in Section 2.2) and the related literature reviewed (and presented in Section 2.3 through to Section 2.5). The DSS development cycle comprises most aspects of the process concepts used in software engineering. It is also true that DSS are to a large extent similar to other software products. For instance, only a small portion of their codes relates to things seen by the end user. Overwhelming amounts of codes concern infra-structural items such as databases, protocols, resource management tools, and other 'hidden' functions, which relate to the developers. Testing by systematically involving various stakeholders (including the developers and even other experts as well), as subjects should be emphasized. Other similarities include resemblance of programming styles and 'fashioned' use of widely accepted programming languages in writing source codes. Despite of these similarities, there are also several significant differences.

One of the unique characteristics of the processes of development of DSS is that they are based on engineering physical phenomena and thus are naturally research oriented. Novel DSS usually originates from concepts or theories, some of which are new and have never existed or been used before. The developers typically work with the eventual software at the design phase under various different guises. The life cycle stages of the in-process implementations at the design phase implicate abstraction levels of the DSS. The levels have, however, no strict boundaries. Transition from one level to another happens intuitively. This special nature of the DSS design process poses a unique software process management and quality assurance challenge.

What makes the development of DSS systems particularly hard is the complexity of the product design process they are supposed to support. Product design is known to be one of the most demanding and complex tasks, sometimes incorporating loads of physical phenomena or engineering principles. Whereas in restricted design domains, such as mechanism design and electrical circuitry design, formal methodologies have been applied successfully, this is not so much the case in domains such as freeform shape design, industrial design and product design in general. In the latter domains, the design spaces are typically huge and not formally manageable. Evidently, the demands for quality in the consumer or engineering products industries are very high, so that these industries are forced to enhance quality of their products, and hence strongly need im-
proved DSS systems. It has been shown that the design phase accounts for a significant number of flaws in software projects, and therefore needs to be put into the spotlight. The activities in this phase unintentionally breed flaws, which, if not detected and eliminated, might ultimately affect the quality and trustworthiness of the anticipated software. The following are some examples of flaws that can be traced back to the design phase.

- There can be unintentional deployment of an inappropriate engineering principle or a physical phenomenon. In most cases, many theories that can be used to solve the same problem. Some of them can be simple, but not accurate enough while others can be complex and more accurate, but computationally intractable. A DSS that ignores one or more essential engineering principles may be quite adequate, but it gives misleading results under other conditions. And since the theories are concealed in the DSS, their inadequacies may not be noticed. Furthermore, often programmers are not fully versed on the adopted foundational theories. On the other hand, the DSS designers who choose theories may also not be skilled enough to notice that the final DSS is a faithful implementation of a right theory or set of theories.

- Reuse of codes or methods\(^3\) is a common phenomenon in software development. When using the existing methods (that have been translated into codes) there is always the danger that inappropriate design decisions (as far as the problem in hand is concerned) entrenched in the codes can be institutionalized. It can also be difficult to adequately synchronize the existing methods to the current design.

- An algorithm can be unstable, and can generate inaccurate values for some inputs. It must be noted that some algorithms allow errors to accumulate, and this can severely distort results. Computers only approximate real numbers and thus numerical errors are inevitable. Two algorithms that are logically equivalent\(^4\) can generate different numerical values when implemented with finite arithmetic.

- During coding, the programmer can, for example, unintentionally exclude special cases; overlook logical steps defined in the algorithms, and so forth.

These kinds of errors are common, and they can be serious. As shown in this chapter, there is a great deal of techniques for assuring quality, but there is no any systematic procedure that addresses the above-mentioned breeds of errors, in particular those that originate in the activities performed prior to coding. It is therefore tempting to suggest that there is a need for a dedicated and meticulous methodology for quality assurance of the preliminary (abstract) implementations at the DSS design interval. The following are some characteristics that such a methodology should embody. It should:

- Suit the processes of development of DSS systems, which are based on physical phenomena or engineering principles. It should in particular systemize the process

---

\(^3\) Codes embody methods.

\(^4\) Logically equivalent algorithms compute the same results with perfect arithmetic.
of designing the DSS. The steps of the DSS design process and the deliverables in each step should be defined.

- Provide a structured approach for prototyping and reviewing various in-process implementations of DSS at the design phase. However, it should not be a monolithic methodology that tries to strictly define each and every activity.

- Provide a way for involving various stakeholders in the development and in verification and validation of the in-process implementations of DSS at the design phase.

- Provide a scheme for testing various in-process implementations to requirements as the design process progresses.

- Allow for consideration of quality characteristics recommended by various quality standards when designing DSS.

- Offer an objective mechanism for measuring how well various in-process products are implemented.

- Provide a framework for acquisition, representation, storage and management of knowledge generated in various stages of the DSS design process. Such knowledge can be used in other projects and it can also be transferred to future-generation developers.

- Be possible to incorporate the methodology in the framework of the conventional software process models. This is to say that the application of the new methodology should not imply the departure from the application of the traditional software development models and V&V strategies.

- Address the interplay of the academia (who traditionally develop engineering theories or methods) and the industry (who converts theories or methods into applications), which is common in the processes of development of DSS.

Such a methodology (introduced in the subsequent chapters) can help the developers create the DSS with the acceptable level of quality. In essence, it has to be a prototyping strategy in which prototypes of various in-process implementations provide the basis for repeated clarification of problems and reviews. The requirements for the in-process implementations at the design phase can be appropriately worded and used as items on a checklist or evaluation criteria when reviewing the in-process implementations.

2.8 Conclusion

A review of the state of the art and practice in the development and testing of DSS has been conducted. Capabilities and limitations of various strategies used have been presented and discussed. As a conclusion, it can be said that the existing development models and testing techniques are to a large extent either too local, partial, or too general, and therefore do not scale to precisely match the needs during the development of
DSS. A more systematic and rigorous methodology is required to ensure effective development and testing of these software tools. All intermediate implementations need to be reviewed or tested, and various stakeholders must systematically be involved in this. Enhancement of the existing development models and testing strategies to create a method that can adequately support detection and elimination of faults in DSS, especially at the design interval, which accounts for a significant number of net faults, is the main goal to achieve. What is needed is a structured prototyping and review framework for the in-process implementations of the DSS at the design phase. The following chapters present an attempt to pursue this goal.
Theoretical Fundamentals of Abstract Prototyping

This chapter lays down the theoretical fundamentals of a novel pre-implementation testing concept, called abstract prototyping (AP). First, the reference model is set up, and the basic AP terms are described. Then, the basic principles on quality assurance of the DSS prior to coding are presented. Finally, it puts in context and elaborates two pivotal elements of AP, namely, (i) requirements engineering, and (ii) management of knowledge about the in-process implementations of DSS.

3.1 Setting the Abstract Prototyping Reference Model

One of the initial steps in the development of manual or computer-based systems or methodologies, as distinguished in the literature [refer to, for instance, van der Drift, (1993)] is the identification or creation of a reference model, which among other things, describes the envisaged system or methodology abstractly. Since no formal and systematic ways for ensuring quality of the in-process implementations of DSS at the design phase are available, one of the initial tasks was therefore to create a reference model. Such a model must provide a clear picture and represent the characteristics on which the process is based, and therefore must help to answer questions about the applicability of the model and its underlying concept in real life. It has to be generic enough, represent real situations, and must cover all aspects related to quality assurance of the in-process implementations.

Figure 3.1 illustrates the understanding on the composition of the DSS in the context of this work. Suppose a DSS is stripped of its auxiliary features\(^1\). Only the core codes and user interfaces remain. The leftover is referred to as a *pilot prototype* or a system of pilot prototypes. If a pilot prototype is 'disrobed' from the codes and user interfaces, the remaining form of implementation is an *algorithm* or a collection of algorithms. And an algorithm without details of flows and processing actions is a *method* or a system of methods. A method with sequencing and information about parameters excluded is a

---

\(^1\) Auxiliary features refer to secondary functions of the DSS such the help, tutorials, etc.
theory or set of theories of an engineering physical phenomenon. Without formalization, the solution left is just chunks of ideas. It can therefore be said that the DSS tools are basically the embodiments of theoretical models of engineering principles or physical phenomena, and the embodiment process passes through four stages, and yields theories, methods, algorithms, and pilot prototypes as the in-process implementations. The embodiment process is preceded by the ideation process, which is essentially an exploration of the possible solutions, while taking into the consideration the specification requirements and solution constraints.

Section 2.2 showed that literature and empirical evidence in the real world vindicate that the converse of the chronology of reasoning represented in Figure 3.1 is what actually happens intuitively in practice. The DSS design process starts by some sort of insightful exploration of solutions to the problems, followed by the concretization process. In the concretization process, the ideas are formalized, making use of proved supporting theories, and based on the theories, the underlying methods are formulated. The algorithms are subsequently designed based on formal methods. As part of the DSS design process, a pilot prototype can also be implemented to clarify the problem or the solution concepts. It is important to mention that these activities in actual practice are not necessarily sequential, and may be carried out recursively as shown in Figure 3.5

Let I, T, M, A, and P denote sets of ideas, theories, methods, algorithms, and pilot prototypes respectively, where: \( I = \{i_1, i_2, \ldots, i_n, i_s\} \); \( T = \{t_1, t_2, \ldots, t_m, t_s\} \); \( M = \{m_1, m_2, \ldots, m_{m-1}, m_n\} \); \( A = \{a_1, a_2, \ldots, a_{a-1}, a_s\} \); and \( P = \{p_1, p_2, \ldots, p_{p-1}, p_s\} \); and \( S \) denote the final DSS product. Let also \( \gamma_{fea}, \gamma_{lan}, \gamma_{str}, \gamma_{fun}, \) and \( \gamma_{for} \), where; \( \gamma_{fea} = \{y\}, \gamma_{lan} = \{-l'\} - \{c\}, \gamma_{str} = \{-f\} - \{a'\}, \gamma_{fun} = \{-s\} - \{p\} \) and \( \gamma_{for} = \{-z\} \) [see Figure 3.1] stand for feature, language, structure, functionality, and formalization filters respectively; \( y, l', c, f, a', s, p, \) and \( z \) being auxiliary features, interfaces, codes, flows, action, sequence, parameters and formalization respectively. A theory, a method, an algorithm, a pilot prototype and a DSS product can therefore be defined in the AP context as follows.
Theory: A formalized representation of principle or practice established by reasoned arguments based on known facts and rules, which formally describe software solution ideas. A theory \( t \) can formally be defined formally as \( t_i = \gamma_{\text{for}}^{-1}(t_i) \), and \( T = \sum_{i=1}^{n} t_i = \{ t_i \} = \gamma_{\text{for}}^{-1}(T) \). This expression means that \( \gamma_{\text{for}}^{-1} \) adds formalism (z) to ideas, and the result is a theory.

Method: A functional representation of the solution concept, with the consideration of the appropriate sequence of theories and organization of parameters from a computation point of view, but without formal structural details or characterization of the solution into a form interpretable by the device\(^2\) to be used for solving the problem. A method \( m \) is hence formally defined as \( m_i = \gamma_{\text{for}}^{-1}(t_i) \), and \( M = \sum_{i=1}^{n} m_i = \{ m_i \} = \gamma_{\text{for}}^{-1}(T) = \gamma_{\text{fun}}^{-1}(\gamma_{\text{for}}^{-1}(T)) \). This expression means that \( \gamma_{\text{fun}}^{-1} \) adds information about sequencing (s) and parameters (p) to theories, and the result is a method.

Algorithm: A structural description of the solution concept into a form interpretable by the device to be used for solving the problem. The description takes into the consideration proper data flows and specification of actions. An algorithm \( a \) is defined in the AP context as \( a_m = \gamma_{\text{for}}^{-1}(m) \), and \( A = \sum_{m=1}^{n} m_m = \{ m_m \} = \gamma_{\text{for}}^{-1}(M) = \gamma_{\text{str}}^{-1}(\gamma_{\text{fun}}^{-1}(\gamma_{\text{for}}^{-1}(T))) \). This expression means that \( \gamma_{\text{str}}^{-1} \) adds information about actions (a') and flows (f) to methods, and the result is an algorithm.

Pilot prototype: An implementation of a specific solution concept, without auxiliary features. A pilot prototype \( p \) is thus formally defined as \( p_n = \gamma_{\text{fun}}^{-1}(p) \), and \( P = \sum_{n=1}^{n} p_n = \{ p_n \} = \gamma_{\text{fun}}^{-1}(A) = \gamma_{\text{fun}}^{-1}(\gamma_{\text{str}}^{-1}(\gamma_{\text{for}}^{-1}(T))) \). This expression means that \( \gamma_{\text{fun}}^{-1} \) adds codes (c) and interfaces (i') to algorithms, and the result is a pilot prototype.

DSS product: A full-fledged implementation of all solution concepts, flanked with all auxiliary features. A DSS product \( S \) is therefore defined in the AP perspective as \( S = \gamma_{\text{for}}^{-1}(P) = \gamma_{\text{for}}^{-1}(\gamma_{\text{str}}^{-1}(\gamma_{\text{fun}}^{-1}(\gamma_{\text{for}}^{-1}(T)))) \). \( \gamma_{\text{for}}^{-1} \) adds auxiliary features (y) to algorithms. In essence, this expres-

\(^2\) The device to be used in solving the problem can be computer, human or both.
sion defines the process of development of the DSS according to the steps introduced in Figure 3.1.

Hence, Figure 3.1 in a way presents a reference model, which forms a framework for prototyping and reviewing the in-process implementations. This reference model defines four hierarchical abstraction levels of the DSS at the design phase, namely theories, methods, algorithms, and pilot prototypes. Theories and methods are the in-process implementations that correspond to conceptualization of a DSS. Algorithms and pilot prototypes correspond to detailed design. Conceptualization in this context involves conceiving how the DSS will be like, while detailed design in this perspective involves refining the conceptual view of the DSS. The issues of concern during conceptualization include creation of theoretical and methodological foundation for the DSS; while in detail design, the issues of concern include specification of the algorithms that implement the functions, definition of concrete data structures, specification of connections among functions and data structures, and architectural design of the DSS. In some cases, the latter extends to writing preliminary codes, which can be run to verify the design concepts or validate the requirements specifications.

It should be noted that strict adherence to the abstraction levels when designing DSS is necessary, and is recommended. However, freedom on the part of the user of the AP concept (that is, the DSS developers) is also permitted in order to accommodate all possible or unpredictable circumstances during the DSS design. For example, in actual practice, the starting point of the DSS design process can be anything, that is, can be a theory, a method, an algorithm or existing codes, and this must be addressed. In fact, theories are tied to methods, to algorithms, to codes that constitute pilot prototypes, and so on. This is to say that theories are the prerequisites in the creation of methods, design of algorithms or in writing codes for functionality. Similarly, whenever there is a pilot prototype (codes) there must be one or more algorithms, methods and theories behind it.

It should be noted that the process of designing the DSS according to the above described reference model naturally consumes time. The developers require time, for instance, to peruse literature or to brainstorm in order to discover sound possible solutions in the form of theories, methods, algorithms or existing codes. From this reality, it follows that it requires time to transform an in-process implementation from one abstraction level to another. It can be said that abstraction decreases if time is available for transformations to take place. And abstraction varies inversely with respect to the transformation duration. In other words, as time is spent in solving a problem, the solution becomes more concrete. Factors such as the skills and knowledge the developer has, the nature of the problem, and so forth influence the transformation time.

### 3.2 Basic Concepts and Terms

As depicted in the previous section, in the process of transforming the requirements for the DSS into software, the anticipated DSS appears in various preliminary forms at the
design phase. DSS appears as an aggregation of supporting theories, a system of methods, a collection of algorithms and finally as a structure of pilot prototypes. These in-process implementations of the DSS are referred to in this thesis as abstract prototypes. The corresponding creation sub-phases are known as the abstraction levels or the stages of the DSS design process. Each level denotes a particular degree of abstraction of DSS, and the levels reflect gradual finalization of the design of the DSS. Typically, there can be many alternative abstract prototypes for the envisaged DSS at the above-mentioned abstraction levels. To develop a quality DSS, the developers must be certain that the abstract prototypes embody the expectations and fulfill their requirements. The process of reviewing the abstract prototypes against the review criteria by involving appropriate stakeholders as subjects is known as abstract prototyping, and is abbreviated as AP throughout this thesis. These review criteria are derived from the requirements. AP is based on the idea that some requirements for the DSS can be verified and/or validated adequately at certain early phases of development. It is intended to improve quality by introducing review cycles at the design interval and consequently helps the developers eliminate faults in the DSS. AP differs from the traditional software prototyping approaches in that it does not assume existence of executable implementations. It involves testing of the requirements for the abstract prototypes of the DSS and predominantly focuses exclusively on code-free verification or validation.

An abstract prototype of the DSS can be represented or modelled just like any other product model, and the resulting representation can serve as a means for communication with subjects or customers. A document describing a purposefully structured set of theories is a DSS prototype. Similarly, a document, graphical representation, and/or audio or video presentation of methods, algorithms or executable codes of pilot prototypes are prototypes of different degrees of abstraction. Like prototypes used in the traditional prototyping processes, the motivation behind using abstract prototypes is economy. Abstract prototypes provide a less expensive, yet concrete basis for discussing difficulties, clarifying problems and for preparing discussions between the developers and other DSS stakeholders early on.

Basically, AP is the symbiosis of a methodology and software tools application in the development and review of the abstract prototypes at the design interval. The purpose is to verify, validate and formally detail the design of DSS. The procedure furnishes the developers with a methodology for exploration and reasoning about alternative solutions at the design phase, and systematically brings into the review process various stakeholders of the DSS, namely, various user groups, developers, and even other experts. It is intended to provide a concept to enable the developers to think beyond their own experiences and expertise, and to reach across various stakeholders and other ex-

---

3 Basically, the review criteria are 'versions' of requirements written for review purposes. They have the same information as the requirements, but couched in a language that can be understood well by the review subjects, i.e. expressed without using formal software engineering terminology.

4 AP can be time consuming and may require additional resources. Nevertheless, the application of AP ensures quality of DSS, and in the long run returns the investments.
Chapter Three

experts, to explore consistency, completeness and validity of solutions to problems using knowledge and experience extracted from them through their involvement as the review subjects. AP is even more than just a methodology. It is a way of thinking that enables the DSS developers to project and reflect the contents of their solutions. It can be considered a post-concept model for evolutionary design methods such as the TRIZ based solution finding [see Altshuller, (1984); Altov, (1994)] and a pre-processor for representation systems [Figure 3.2].

AP comprises reviews that treat the DSS at two levels of complexity, namely, (i) components level, which involves reviewing the abstract prototypes that relate to parts of the DSS, and (ii) assembly level, which involves putting together the abstract prototypes of the components of the imagined DSS system and reviewing the 'integrated' system or sub-system to assure compatibility. Both reviews deal with the intended DSS without the need to construct and install the actual executable prototype.

The aims of AP can be summarized as to:

- Assess the abstract prototypes to investigate how well they have been implemented.
- Formalize the design specification for DSS in order to avoid ambiguity, achieve consistency and attain the required depth in the exploration and crystallization.
- Find the connections and the relationships among various abstract prototypes.
- Measure adequacy and estimate the levels of fulfillment of various desirable quality characteristics such as robustness, reliability, efficiency, usability and convertibility of the abstract prototypes.
- Check how effective the initial stage abstract prototypes such as theories and methods can be used from the later stage abstract prototypes' perspectives.
- Select the most appropriate candidate(s) amongst the alternative solutions.
- Gain knowledge on what and how to improve the abstract prototypes.

Figure 3.2: Position of AP in the DSS development process
3.3 Abstraction Levels

The AP concept is based on the hypothesis that verification or validation can be done successfully as early as when the first tangible solution element starts to emerge, and in various abstraction levels. The notion abstraction level spontaneously serves many purposes. For instance, it implies an order for requirements processing and thus provides, for example, a way to manage which quality characteristics and requirements to consider at various DSS design stages. Requirements are grouped according to the abstraction levels in which it can be checked if they have been fulfilled satisfactorily. It is hypothesized that certain requirements can be tested at certain design stages, and at the same time, there may be some sort of overlaps of the requirements in various stages. Also, coupling of the requirements to the abstraction levels provides a way for testing their fulfillment at the earliest moment, and perhaps at the lowest cost. In this way, the direction of the development process is assured early on, and as a result, the development time, as well as the amounts of rework, can significantly be reduced. Clustering requirements to the abstraction levels [Figure 3.3] is one of the cornerstones of AP. The abstraction levels also provide a way for systematically involving the DSS stakeholders in AP [refer to Section 3.4 and Figure 3.5]5. The remaining parts of this section further elaborate the four abstraction levels of AP.

Theories Level

The entry point in AP is the review of foundational theories for a DSS. In the context of AP, theories are general principles (e.g. physical laws, mathematical and geometric rules, etc.), or practice established by reasoned arguments based on known facts. A theory can cover the entire DSS or a part of it. On the other hand, many theories can cover only a small portion of a function. What is considered a 'theory' can be as wide as, for instance, 'electromagnetism' or as narrow as 'strength of this screw', and must have been validated well before. Knowledge derived from various sources such as from the literature or past experiences provide the basis for formulation of novel

---

5 In this case, compliance of the eventual DSS with certain quality characteristics is evaluated by the users, experts or other stakeholders at certain specific abstraction levels.
theories. Theories can formally be presented in the forms of text and/or mathematical equations, or in graphical, audio or video forms. The purpose of the theory-level AP is to investigate the compliance of the existing or formulated theories with quality characteristics such as completeness, consistency, usability, reliability and maintainability; with respect to the problems that they are supposed to solve. One advantage of theory-level AP is that the risks posed by the present trends, in which decisions on the foundational theories to deploy are somewhat subjective (i.e. highly influenced by emotions or individualistic feelings rather than the merits of the theories), can be reduced through a consensus of the developers and the subjects. It aims at avoiding the risks of adopting theories without thorough assessment. The use of verified theories ensures that the DSS development process proceeds on the right track, right from the early stages.

Methods Level

Theories imply a certain method or a system of methods. A method represents sequencing of the related theories and the parameters required for processing the theories. A method-level prototype should therefore give insight into sequencing of the theories, with the view of attaining optimal flow of processes. It follows that the methods-level AP basically aims at finding the best method or set of methods. When methods are reviewed, the best sets of descriptive parameters are also decided upon. Other issues dealt with include investigation of the necessary inputs and outputs, as well as the storage possibilities.

Algorithms Level

An algorithm is a characterization of a method for solving a problem in a form interpretable by the device to be used for solving the problem, which can be humans, machines or both [Korfhage, 2000]. It is recognized in software engineering that in solving a problem, there can be many algorithms (and consequently programs) to solve it. Not all of them are of equal quality. Typical criteria by which quality of an algorithm is judged include: time and memory requirements, accuracy of the eventual solutions, and generality. An algorithms-level AP involves usage of criteria of these kinds as yardsticks in examining the flow and actions, as well as in judging the consistency and completeness of algorithms. Fulfillment of the ‘algorithmic’ requirements is optimized, and details of features in the DSS such as inputs, outputs, user interface and databases are concretely specified. Also, algorithms give insights into managerial and technical issues such as the amount of effort required to prepare working code, what kinds of resources will be required, and what will be the capability of the imagined system.

6 The term ‘method’ holds the same meaning as defined in the literature, i.e., a method is ‘…a way or manner of doing…’ or ‘…an orderly system or arrangement…’ or ‘…a hidden system behind disordered actions…’ [Longman Dictionary of Contemporary English, 1984]; ‘…a structured set of steps and tasks that have to be carried out…’ [Hlupic and Paul, 1996]; and is ‘…a very useful description of what is to be done next, who is to do it, and how…’ [Avgou and Cornford, 1993].
Pilot Prototypes Level

Pilot prototypes are the executable core functions of the anticipated DSS (i.e. limited implementation of the DSS). Only the algorithms for the core features of the software are implemented at this abstraction level. Typically, there can be several kinds of programs (that is, executable codes) written for a particular problem, but not all of them are of the same quality. The objective of pilot-prototypes-level AP is therefore to find out, by involving representatives of various DSS stakeholders, whether the executable implementations of the most critical parts of the DSS fulfill the functional as well as non-functional requirements.

3.4 I/O of Abstract Prototyping

In order to determine how well an abstract prototype at the design phase has been developed, the review criteria and its representation\(^7\) are the main ingredients needed. Requirements describe the needs, and they are the sources of review criteria. On the other hand, the representation presents the solution concepts, and it might give the look and/or the feel of the abstract prototypes. In essence, the representation can appropriately characterize and reflect features of the DSS. Therefore, to conduct an AP review, (a) requirements, from which the review criteria are derived, and (b) a representation of the abstract prototype, must be formally and appropriately presented. The latter is crafted and tested against the review criteria. The developers or testers oversee the AP proceedings while the subjects provide quantitative values used in the determination of the acceptability and the extent of fulfillment of the review criteria (which is by default the measurement of the fulfillment of the requirements). Figure 3.4 summarizes these ideas and presents the inputs and outputs of AP.

3.5 Role of Users and Developers

AP is a stakeholders-oriented procedure in the sense that the representatives of the DSS stakeholders are systematically involved as evaluation subjects in the review of the abstract prototypes. In the context of this work, a stakeholder is an individual or a company which is or likely to be significantly affected or influenced by the envisaged DSS. Two broad groups of stakeholders can be identified, namely (i) developers and (ii) users. The developers include members of the DSS development team (such as the programmers, testers and managers) as well as experts from other related fields, from within or outside the company.

The users are clients of the DSS, mainly the design companies, in which there are various kinds of users, including the end-users such as draftsmen and design engineers, and peripheral users such as process planners. Harmonious consideration of the views of the stakeholders can contribute to the improvement of quality of DSS. For example, while

\(^7\) Refer to Section 4.2.2 for details on representations of the in-process implementations.
the developers' opinions help build the DSS according to the prevailing advances of the underlying technologies, the users' views let the developers know the state-of-the-art of the needs. The advantage here is that this brings into the design process and the review floor the expertise of the developers as well as the expectations and knowledge of other stakeholders, including the end-users. Figure 3.5 shows the proposed users-developers involvement scheme. Under this scheme both the users and the developers systematically serve as the evaluation subjects. This scheme recommends review of (i) theories by the developers, followed by the revision of (ii) methods by the users; (iii) algorithms by the developers; and (iv) pilot prototypes by the users.

The users (from the industry) can only be involved in reviewing the abstract prototypes that they can easily perceive, namely (i) less technical methods\(^\text{9}\) and (ii) pilot prototypes. On the other hand, the developers (from the academia and software development firms) are naturally technology-minded, and can serve as subjects in the review of theories and algorithms, as well as the highly technical methods. For a successful review, the methods for representation of the abstract prototypes in perceivable forms must be

---

\(^8\) Highly technical methods should, however, be reviewed by the developers.

\(^9\) Some methods are highly technical, and it may require significant efforts to let third party individuals sufficiently perceive them and provide meaningful review. On the other hand, others are less technical such that even novice individuals can understand them.
Theoretical Fundamentals of Abstract Prototyping 53

devised, and the representations themselves must effectively help invoke comments and suggestions from the subjects. The subjects, in particular the end-users, are arbiters of the DSS system being developed; they know what will work and what will meet their demands. Their involvement in reviewing concepts and solution proposals strengthens the chance of acceptance of new computer-based theories and methods. In this way, the links between the industry (that uses the DSS) and the academia (that traditionally finds the underlying theories and methods) can also, by default, be established early on. It can be said that this scheme serves the purpose of helping the developers reconcile the needs and the requirements from various sources. In addition, it provides a framework for co-designing the DSS with its stakeholders, by involving them in refining and in extending the initial product concepts, often put forward by the developers.

Representation and processing of the requirements as well as the information about the abstract prototypes are the pivotal constituents of AP. The following sections present the principles upon which these activities are based.

3.6 Requirements Engineering in the Perspective of the Abstract Prototyping Concept

Quality is defined as conformance to requirements [Crosby, 1996]. This implies that in order to develop high quality abstract prototypes, their requirements must be comprehensively defined. This paves the way for development of acceptable intermediate products. As pointed out in the literature, for instance, in Fezza et al, (1996) and in Jones, (1990), requirements describe the objectives and capabilities of the intended software. Requirements often fall into two categories, namely: for those who commission, pay for, or use software; and for those who write the codes [see e.g. Gunter et al, 2000].

AP is basically a requirements-driven quality assurance procedure. In the context of AP, the contemporary definition of the term 'requirement' is upheld, but focus is on the requirements for the abstract prototypes at the design phase rather than on requirements for the (entire) intended end product (i.e. software system). It is known that different problems bring different requirements into focus [see e.g. Carroll et al, 1995]. Similarly, functions or features in different stages of development associate to different re-
requirements. Like for the 'system' requirements, the 'level-wise' requirements must address what problems at a given level need to be solved (not how to be solved). These requirements ultimately provide the focus for implementation of the abstract prototypes. The mere existence of the 'level-wise' requirements in a way clarifies the DSS project's goals at a given design stage. These requirements must, like the 'system' requirements, address the needs of various stakeholders. In fact, they are also the statements on the capabilities required by the stakeholders, and they should not be obscured by proposed solutions. Typically, various stakeholders often have different demands on what the eventual DSS must do for them, and considering their requirements can give rise to an acceptable abstract prototype. One of the challenges is how to acquire good requirements for the abstract prototypes that would ultimately ensure quality of the eventual DSS; and how to represent, store, and reuse them.

3.6.1 Fundamentals

This section addresses the challenge of handling requirements for AP purposes. It describes the characteristics and dependencies of requirements, and explains how the relevance of the requirements and the uncertainty of the review criteria can be handled.

Characteristics and Dependencies of Requirements

One of the primary goals of AP is to ensure that functional and non-functional requirements for the eventual DSS are gradually met at various stages of the design process. Consequently, reviews need to be performed at all abstraction levels, as this would ensure repeated consideration of as many requirements as possible, and will also ensure consideration of the emerging requirements. For a thorough and meaningful review, it is important to keep an eye on proper formulation of the requirements as well as on the relationships among the requirements\(^{10}\) in the requirements model\(^{11}\).

Empirical analysis of the requirements for the DSS show that a requirement or pair of requirements can exhibit the following characteristics: branching, childhood, contradiction, parallelism, relaxation, and/or transitivity.

**Branching:** A requirement is referred to as a 'branching' or a 'parent' requirement if one or more requirements descend from it. Such a requirement is broad and rather general requirement, and can be split into smaller, more specific child requirements that share global context of the need(s) it addresses.

**Childhood:** A 'child' requirement descends from a 'branching' or 'parent' requirement. Fulfillment of a 'child' requirement implies partial fulfillment of its parent requirement.

---

\(^{10}\) The term 'requirements' holds its state of the art definition, i.e., requirements are descriptions of what software will do [refer to Section 2.5], and they essentially express wishes and conditions of various stakeholders, as well as technological constraints.
Contradiction: Contradiction arises if two or more requirements address mutually exclusive and opposite needs. Improvement of an abstract prototype to fulfill one of two contradicting requirements leads to deterioration of the fulfillment of the other. Fulfillment of the highly weighted requirements, or in other words, the highly valuable requirements (as far as the clients are concerned) would be one of the possible remedies for the contradiction problem.

Parallelism: Two requirements are said to be parallel if they echo similar needs. Improvement of an abstract prototype to fulfill one of the parallel requirements results into automatic fulfillment of the other.

Relaxation: A requirement is said to be relaxed if it is not affecting other requirements and at the same time it is unaffected by the rest of requirements. A relaxed requirement is generally a less constraining requirement, and points to a unique goal. Such a requirement addresses distinct needs. Improvement of an abstract prototype to fulfill such a requirement will not have influence on other requirements.

Transitivity: A requirement is said to be transitive if it is part of a chain of requirements that depend on, or are tied to each other, such that fulfillment of one requirement alone would not be enough. Fulfillment of a transitive requirement is conditional for complete satisfaction of the needs it addresses.

Considering their actual contexts or implications, requirements might or can strongly relate to each other. The relationships can be described as web, typically with many-to-many (n x m) dependencies (Figure 3.6). Dependencies among the requirements extend beyond the abstraction level. Some requirements can address the needs in more than one abstraction level (that is, \( I \times n \) relationship) while in other cases; requirements address the needs in one abstraction level only (that is, \( I \times I \) relationship). This is to say that, in AP, there are requirements that are valid in one abstraction level only, while some requirements are valid in more than one level.

Figure 3.7 shows a set representation of the relationships among the requirements. If \( R^k \), \( M^k \), \( A^k \), and \( P^k \) are sets of requirements for the anticipated DSS at theories, methods, algorithms and pilot

---

11 The term 'requirements model' is used in this thesis to refer to a complete list of requirements at a given abstraction level. Refer to Section 3.6.3 for more details.
prototypes abstraction levels respectively, then set $C^R$, whereby, $C^R = T^R \cap M^R \cap A^R \cap P^R$ represents common requirements. $C^R$ comprises requirements that must be fulfilled at all four abstraction levels. Similarly $T^R \cap M^R$ corresponds to requirements that must be fulfilled at the theories and methods levels, $M^R \cap A^R$ represents requirements that must be fulfilled at methods and algorithms levels and so forth.

**Classification of Requirements**

There are numerous ways to distinguish requirements. For the sake of AP, apart from the abstraction levels in which they can be tested, requirements can also be grouped according to quality characteristics they represent. Requirements also have different sources, can represent different needs, can describe different functionality, and can be specific or generic (as far as their application scopes are concerned).

**Specific and Generic Requirements**

Some requirements describe the characteristics of more than one function of a DSS. These are referred to in this work as *generic or general-purpose* requirements. Generic requirements are valid and address the needs in more than one function. On the other hand, there are requirements that relate to only a singular function. These are known as *specific requirements*. Specific requirements are functionality specific, and in most cases they must be fulfilled unconditionally. Both generic and specific requirements serve as the sources of the review criteria used in AP.

**Functional, Non-Functional and Quality Requirements**

A requirements model\(^{12}\) typically consists of functional and non-functional requirements. Functional requirements specify tasks or behaviors that the envisioned DSS must support or address, while non-functional requirements describe additional quality features that must be accounted for in order to create an acceptable DSS. The former

---

\(^{12}\) Refer to Section 3.6.3 for the description of what a requirement model is.
requirements must be fulfilled as completely as possible, and as early as from the theories abstraction level.

Functional requirements $F_r$ form an essential part of quality requirements $Q_r$. Practically, functional requirements are subsets of quality requirements, that is, $F_r \subset Q_r$. [Figure 3.8]. This is also true for non-functional requirements $NF_r$, which are also part and parcel of quality requirements. This means that both $F_r$ and $NF_r$ serve the roles of $Q_r$. For example, in terms of the ISO/IEC 9126 quality standard, functional and non-functional characteristics such as efficiency, functionality, maintainability, portability and usability constitute quality characteristics. This is to say that fulfillment of quality requirements implies fulfillment of both functional and non-functional requirements.

As far as this work is concerned, quality requirements are essentially the yardsticks of the acceptability of the envisaged DSS and of the abstract prototypes. However, it should be noted that some of the quality requirements are often more valuable than the others, and the DSS design process can therefore be given a go ahead, beyond a given abstraction level, if the key requirements, for instance, the functional requirements are deemed to be satisfied well enough\textsuperscript{13}. To reduce complexity when assessing compliance of an abstract prototype with the requirements, it is recommended that each requirement must be dealt with independently.

**Relevance of Requirements**

The significance of each requirement must be specified. Typically, there are requirements that: (i) must be fulfilled unconditionally; (ii) must be fulfilled as completely as possible; and (iii) are rather unimportant and whose fulfillment can only be insisted. Both the subjects involved in AP and the developers can be involved in ranking of requirements and classifying them in these groups as follows. First, the subjects have to indicate the relevance ($\rho$) of each requirement, and the developers can subsequently consider these values as the basis for assigning weighing factors ($\omega$) to every requirement\textsuperscript{14}. The weighting factors in turn form the basis for classification of requirements. In this way, both the subjects (i.e. the representatives of the stakeholders) and the developers bear the responsibility of deciding about in which category each requirement

---

\textsuperscript{13} This implies a subjective factor in that different people in the development team can judge fulfillment of requirements differently. There are several possible remedies to this, one of them being to ensure that all functional requirements are approximately satisfied to the same level.

\textsuperscript{14} Refer to Section 4.3.2 and Appendix D for further information about $\rho$ and $\omega$. 

belongs to. These categories have implication on the extent to which each requirement must be met by the abstract prototype. It is important, however, to mention that categorization of requirements in the order of importance is never a straightforward process, since $\rho$ and $\omega$ values, typically vary with the problem. They merely express judgments of some individuals, and can therefore naturally be subjective. Experience and skills are valuable indeed in this pursuit. As for treatment of the above-mentioned categories of requirements as far as using them in AP is concerned, the following is proposed. The requirements that must be fulfilled unconditionally should either be omitted from the list of requirements to be paraphrased and used as review criteria or heavier weights should be assigned against them. This is to say that during the AP reviews [see Sections 4.1.3 and 4.2.3], the information gathering matrix [refer to Section D.2 in Appendix D] should either contain all three categories of requirements (in this case, the requirements must be weighted accordingly), or those that must be fulfilled unconditionally should be excluded from the information gathering matrix.

**Validity of Review Criteria**

The DSS developers also have the responsibility of transforming requirements into review criteria. This involves paraphrasing the requirement statements using language that can be understood well by the targeted subject. On the other hand, in AP, subjects are required to understand precisely the semantics of the review criteria. It is uncertain whether the review criteria can be formulated as precise as they should be. On the other hand, it is also uncertain whether the subjects would understand the review criteria as clear as the developers would wish. It is therefore necessary to investigate if a review criterion is valid, that is, if it is well formulated and understandable to the subjects.

In the validation of the review criteria, two types of weights [Figure 3.9], one assigned by the developers and another assigned by the subjects, are proposed to provide some sort of consensus. Weights assigned by the developers ($w_d$) take care of the uncertainty on the side of the developers, while weights assigned by the subjects ($w_s$) take into account the uncertainty of the subjects. The decision on the validity of the individual review criteria should therefore be carried out.

![Figure 3.9: Handling uncertainty of review criteria](image-url)
based on the consensus weights $w_c$; where $w_c = \sqrt{w_d w_s}$\textsuperscript{15}. A review criterion can thus be considered as well formulated if it scores at least a pre-established threshold consensus weight, $w_{th}$. The value of $w_{th}$ can be set, say, based on experience or empirical evidences. This can help the developers remedy the uncertainty problem and achieve reasonably precise and reliable review criteria for AP.

### 3.6.2 Acquisition of Requirements

Requirements acquisition is an activity performed prior to commencement of the DSS design activities. Usually, given a concise statement of the problem and an indication of constraints that exists for its solutions, requirements can be formulated. Figure 3.10 shows typical sources of the requirements used in AP. Requirements can be derived from marketing documents, expert knowledge, business rules, past experiences, experimentation with prototypes, analogies, analysis of existing solutions, based on knowledge derived from the literature, and industry standards. Requirements used in AP can also be a result of an intuitive imagination and reasoning, typical in brainstorming, for instance with experts from the relevant fields, practitioners, and other DSS stakeholders. In general, requirements formulation and validation can be considered as a recursive loop of the developers-stakeholders-experts consultations in which requirements are proposed and improved (i.e. analyzed, visualized, modified) repeatedly.

It is important to keep track of all requirements and to ensure that they are correctly acquired from their sources. The starting points in the process of acquisition of requirements for AP are the proposed DSS features or functions; and the roots of requirements are quality characteristics. The desired quality characteristics must thus be identified first. Various software quality standards describe the basic characteristics to be considered in the evaluation of software. Most often, testing to requirements that belong to quality characteristics derived from standards can be done fully after the implementation of the entire system or partly at the early development phases.

### 3.6.3 Requirements Model

A list or document of requirements describing quality characteristics for an abstract prototype and providing additional connectivity information about the individual re-

\textsuperscript{15} In case of extremely deviating values of $w_d$ and $w_s$, a simple average of the weights provides an unrealistically large $w_c$ value. The square root of the product of $w_d$ and $w_s$ provides a more realistic $w_c$ value.
requirements is referred to in this thesis as a requirements model. Standards give the developers a good start in formulation of the requirements models. Preference must be given to quality characteristics recommended by standards such as ISO, IEEE and IEC [Figure 3.11]. Such quality standards define essential characteristics that must be taken into account when compiling requirements for use in AP. For example, according to the ISO/IEC 9126 standard, quality is defined in terms of functionality, reliability, efficiency, portability, usability and maintainability. One of the advantages of using standard quality characteristics to describe DSS quality is that the requirements can be formul-

![Diagram](image)

**Figure 3.11: Organization of requirements according to quality characteristics and abstraction levels**

ulated in a way that leaves no room for ambiguity. In this way, incompleteness or inconsistency of requirements when using standard norms can generally be minimal. Characteristics other than those affiliated to standards such as security and survivability can be incorporated as well.

Quality characteristics can be broken down into more granulated sub-characteristics. This helps in the clarification, in the finest context, of what is required. They can also be tallied according to the abstraction levels, although there may, however, be some degree of overlapping of the characteristics among the abstraction levels. This allows the developers to test the compliance of the abstract prototypes with the specific quality characteristics. For each characteristic, one or more requirement statements can be formulated. Information about the sources of requirements, stakeholders they relate to, and so forth, can subsequently be assigned to each requirement statement.
3.7 Knowledge Management in Abstract Prototyping

There are several perceptions on the use of the terms 'knowledge, information and data'. The general consensus is that data are symbolic surrogates used to represent entities; information is the meaning of data, and is obtained through interpretations, abstractions or associations of data; and knowledge is a result of experience with information, or simply putting the information into action [refer, e.g. to Blumenthal (1969); Burch et al (1979); Dretske (1981); McMahon and Lowe, (2002), Owen and Horváth, (2002); Wilson, (1987)]. In the context of the AP concept, information about the abstract prototypes at the design phase, in particular technical information, yields knowledge, which forms the basis for developing DSS.

As pointed out in Section 2.5.3, a software development process is also regarded as a knowledge acquisition process. This is also the case in the processes of development of DSS. Figure 3.12 illustrates how knowledge about the solutions to problems matures as the DSS development progresses. Knowledge matures at a faster rate at the design interval (which is also the AP interval) and the maturity rate decreases steadily afterwards. Such knowledge is based on information such as what theory or set of theories can best be used in the development of functionality, what design of algorithm is suitable, which existing source codes can be reused, and so on.

Such information can be structured appropriately, represented and stored. In fact, storing every kind of information that pops up during development is a very healthy practice, but also a big challenge. Such information can be reused in future DSS development projects\(^\text{16}\). It is generally acknowledged that reuse of information is a well-suited strategy for shortening the development time and cutting the development costs.

\[\text{Figure 3.12: Maturity of knowledge about the solutions to a problem}\]

\[^{16}\text{The ideas considered to be null and void can later be deemed useful. This is typical in practice, where the ideas that pops up in the early phases (such as theories, existing codes, and so on) and rejected for some reasons are often recalled and reuse during reworks.}\]
3.7.1 Representation of Knowledge

The representations of theories, methods, algorithms and pilot prototypes provide information or knowledge about the envisaged DSS in various contexts. One of the important issues in AP is to make the subjects and the developers well-versed about the abstract prototypes. Thus, there is a challenge of letting the subjects, including rather naïve subjects from outside the development team, be well-informed about the abstract prototypes and the concepts behind them. This requires the abstract prototypes in question to be represented and made available to subjects, in any geographical location, in a precise and expressive way.

Like in the domain of systems analysis and design, formal representations of the abstract prototypes at the design phase can typically be utilized for two main purposes, namely to: (i) more succinctly describe features of the abstract prototypes, and (ii) reduce cognitive load by serving as communication media. The subjects should be able to understand and interpret the representations (i.e. data models of the abstract prototypes) and map formal notations, for instance, mock-ups, storyboards and so forth to the reality. The easiness of understanding the representations is the measure of the usability of the representation. The importance of making the participants of an AP review to become well informed about the abstract prototypes is pretty evident. Having well-informed subjects generally implies proper judgment during AP. The more proper the judgment, the lower the chance of faults passing into the subsequent stages.

The abstract prototypes can be represented for AP purposes pictorially, symbolically, linguistically, algorithmically or virtually\(^\text{17}\). The specific representation methods proposed for the abstract prototypes are presented in Section 4.2.2. In order to provide more information about some of the abstract prototypes, in particular algorithms or pilot prototypes can also be made available in the form of executable codes. With executable codes, some sort of hands on tests can be carried out to gain insights into, for instance, the usability of an implementation or algorithm, its efficiency, reliability, and so on. In general terms, the decision on the type of the representation method to adopt must take into the consideration the nature of information to be represented. The representation method selected to represent the information must, among other things, allow for the information to be manipulated and communicated effectively.

3.7.2 Relationships among Abstract Prototypes

There are vivid ties among the abstract prototypes within and beyond the abstraction levels. Figure 3.13 illustrates the complex 'vertical' and 'horizontal' relationships that exist. Typically, there are connections, for instance, among the abstract prototypes at higher abstraction levels and those in the subsequent levels. The use of a certain theory or set of theories can imply automatic use of other theories, certain methods, algorithms or even certain existing codes. Similarly, the use of a certain method can lead to auto-

\(^{17}\) These are indeed the classes of the knowledge representation methods as suggested in the literature. Refer, for instance to Owen and Horváth, (2002).
matic adoption of a certain theory or set of theories, and the use of certain algorithms implies embracement of certain theories and methods. The choice of certain existing codes means 'involuntary' deployment of certain theories, methods and algorithms, which the codes are based upon.

An account above shows that it possible to find out related solutions. This is to say that when looking for the existing solutions for a well-defined problem with well-known needs or requirements, their connections with other solutions within the abstraction level and beyond can be traced. For example, suppose a theory has been chosen and the developers want to know what other known solutions must be used. This can be achieved as follows. Given, say, a theory $t_i; t_i \in T$, where; $T$ is a set of theories for a given functionality, sets of methods $M' = \{m_1, m_2, ..., m_e\}$, algorithms $A' = \{a_1, a_2, ..., a_j\}$, and pilot prototypes $I' = \{i_1, i_2, ..., i_z\}$ that match the problem can be discovered or found, provided that the needs (or the key requirements that must be satisfied unconditionally at all abstraction levels) are known. Such relationships among the abstract prototypes can be formalized and computerized such that it could be possible to investigate the implications of the selections made. For instance, if a theory is chosen, it would be possible to find out which other theories must be used, what methods and algorithms must be adopted, and what existing codes can be used during the implementation of pilot prototypes, and so on. In essence, the relationships among the abstract prototypes formulate a sequence of mappings, which specifies, for example, if a theory $t_i$ is used, then theory $t_j$ (and perhaps also theory $t_k$, and so on) as well as a method $m_y$, an algorithm $a_z$ and so forth must be used. Similarly, a sequence of mappings for counter-indications can also be derived, whereby if, for example, an algorithm

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{abstract_prototypes.png}
\caption{Network of abstract prototypes}
\end{figure}
$a_i$ has been selected, then it means that a theory $t_i$, theory $t_j$ (and perhaps also theory $t_k$, and so on), as well as a method $m_j$, and so forth have been chosen, by default.

### 3.8 Conclusion

The theoretical fundamentals of AP have been articulated in this chapter. A reference model for AP has been defined, the basic terms have been described, and various theoretical concepts related to quality assurance of the DSS prior to coding have been introduced. Primarily, AP can be described as a prototyping and review concept. It is intended for use in quality assurance of the intermediate implementations of the DSS at the design phase, namely theories, methods, algorithms and pilot prototypes. These intermediate implementations have been named abstract prototypes. AP can also be aptly described as an auditing tool for verification of abstract prototypes, and it provides means for checking what quality characteristics to consider in ensuring that they are well-realized and that they are passed into the subsequent stages with the expected level of quality. The emphasis in AP is on code-free verification or validation. The goal is to detect and eliminate faults in the abstract prototypes by systematically seeking opinions of various DSS stakeholders, who are involved as the review subjects. AP differs from traditional verification and validation strategies in that: (i) reviews are predominantly performed prior to coding, (ii) the representations of the abstract prototypes provide means for clarifying and discussing the solution concepts, (iii) the representatives of the DSS stakeholders are systematically involved in the reviews as the design process progresses, and (iv) requirements provide the basis for assessment and consequently quantifying the acceptability of the abstract prototypes. The following chapter elaborates the strategies for deployment of the AP concepts introduced in this chapter.
Abstract Prototyping Process

This chapter gives an account of the AP process. The AP activities are specified, and formal methods for guiding the AP activities and for involving various stakeholders in AP are presented. The procedure for gathering opinions of various DSS stakeholders is presented as well. It then describes in detail the activities supported by computers, namely requirements processing, knowledge management, and information analysis.

4.1 Application Methodology

This section presents the procedural issues related to putting the AP concept in use, and describes various automated and manual activities involved. The problem underscored and dealt with is that the AP concept is novel, and a formal AP methodology is not available.

4.1.1 Abstract Prototyping Activities

Chapter 3 provided clues on the activities involved in AP. This section presents in detail the AP activities. Figure 4.1 gives an overview of the AP process. It shows that in AP, the abstract prototypes are reviewed against the review criteria derived from the requirements. This is done in four abstraction levels, starting from the highest abstraction level, namely, the theories level, followed by methods, algorithms and finally pilot prototypes level. AP is preceded by requirements specification, and is followed by implementation and testing, which involves writing codes for all algorithms of the imagined system, integration, and conducting the traditional software tests such as system and integration tests. Tests on a full-scale prototype help to determine the compliance with the global specifications requirements, which cannot be dealt with in the AP interval. It must be noted that analysis and review of critical functions can be done at the pilot prototypes level, and does not have to wait for a full-scale implementation.

AP can proceed sequentially or iteratively, depending on whether or not further improvements are needed in various abstraction levels. At each abstraction level, the solution scenarios are evaluated and the best ones selected according to the laid down procedures described in Section 4.2.3 and in Appendix D.
It is important to emphasize that in order to effectively review an abstract prototype, the availability of its representation is vital. Also, when testing at an abstraction level in which both the users and the developers are involved as subjects, the review criteria representing singular sets of requirements must be used. The review criteria must, however, be stated in a form understandable to the targeted subjects. AP consists of activities that can be broadly grouped as (i) front-end, and (ii) downstream activities [Figure 4.2]. The front-end activities include all activities related to preparations for the implementation of AP while downstream activities include field activities, data analysis, and interpretation of the results.

![Diagram of Abstract Prototyping Process]

Figure 4.1: Abstract prototyping process

---

1 The necessity of addressing software engineering issues or concepts in the 'language of the targeted audience' is recognized by researchers and practitioners. Refer for instance to Glass, (2002).
Front-end AP Activities

The front-end activities are those performed prior to commencement of AP, mainly to prepare the inputs of the downstream AP activities such as review criteria and representations of the abstract prototypes. One of the initial tasks is preliminary analysis of the DSS project. This involves exploration of the needs and the goals, as well as acquisition of requirements for the abstract prototypes. Such an undertaking can be based upon strategies such as brainstorming, literature reviews and interviewing spectrums of stakeholders. This enables the developers to clarify the problem, identify difficulties and discover solution strategies such as candidate foundational theories or methods, existing algorithms and even available codes. It also helps the developers synthesize the scattered solutions into a common framework. The candidate solutions are later reviewed in the framework of the AP concept to determine whether they can be adopted. The collected requirements must be specific, measurable, and realistic, and must address the needs of the DSS stakeholders. The DSS project goals can be placed within a specific timeframe, and must be in line with, and contribute towards satisfying the stakeholders' needs harmoniously.
In addition, the state of the art of the affairs must to be studied to gain comprehensive knowledge of the situation. Focus should not only be on the understanding of the problem, but also on the exploration of the consequences of the possible solutions. Other preparation activities include: (i) identification and selection of method of representation of the abstract prototypes; (ii) selection of subjects; and (iii) designing demonstrations or assignments to be used in reviews.

**Downstream AP Activities**

The downstream AP activities include: (i) reviewing the abstract prototypes to determine the extent to which the requirements are being fulfilled; (ii) comparing alternative solutions, ranking them, and selecting the most appropriate ones; and (iii) enhancing the selected alternative(s). Requirements for the DSS or its elements formulated previously\(^2\) are sorted out and a provisional requirements model for the abstract prototype in question is retrieved, the requirement statements are validated and rephrased using terminology that the targeted subjects can understand, and finally used as the review criteria. The review criteria for the functionality under consideration are forged into an appropriate format and eventually used in the reviews. As mentioned in Chapter 3, the review subjects can be either the developers or the users; depending on which abstraction level the review is carried out.

**Relationships of the AP Events**

Figure 4.3 shows the relationships of the events in AP. For a given DSS feature, specific needs of the stakeholders, as well as the implementation constraints are established and requirements formulated and classified according to the abstraction levels. Then the AP methodology provides the procedures for prototyping and reviewing the abstract prototypes. The outcome of the AP process is an optimum abstract prototype that has been verified by involving appropriate stakeholders and deemed to satisfy requirements to an adequate level\(^3\). If there are several solution proposals, then they must in addition be arranged in the order of salience.

---

\(^2\) Like in typical software projects, requirements acquisition in the development of DSS precedes the design process. This is to say that requirements for AP are acquired as well during the needs analysis and requirements specification phase.

\(^3\) A decision on whether or not a requirement has been satisfied to an adequate level should be a consensus of the developers and the subjects involved in AP. Formal or empirical statistical methods can provide reasonable quantitative values, upon which the decision can be based.
4.1.2 Driving Schemes

AP is driven by two schemes, which help systemize and direct various activities. These are: (i) the intra-abstraction level improvement scheme; and (ii) the stakeholders' involvement scheme. The AP reference model presented in Figure 3.1 and elaborated in Figure 4.1 provides higher-level guidance of the AP activities. Reviews in this framework proceed recursively among the abstraction levels. The recursive approach is considered to be appropriate because it embodies the evolutionary nature of the DSS design processes and allows the developers to get feedback of activities. The abstract prototypes of the whole DSS system as well as of its elements can be reviewed in this framework.

The Intra Abstraction Level Improvement Scheme

The intra abstraction level review, selection and improvement scheme [Figure 4.4] systemizes the AP activities within the abstraction levels. There are two main tasks, namely, (i) reviewing, ranking and selection of solution proposals, and (ii) improvement of the selected solution(s). This model provides a roadmap through which the solution scenarios in the form of theories, methods, algorithms or pilot prototypes can be reviewed against the review criteria, and better solutions selected. The selected solutions are subsequently analyzed, visualized and modified recursively until the review criteria (and by default the requirements) are satisfied sufficiently.

The analysis focuses on the investigation of the (positive and negative) consequences of the scenario from users' or developers' perspectives. Visualization involves enhancement of the positive consequences and mitigation of the negative consequences, while modification is the actual implementation of the new ideas to create an improved scenario. Chapter 6 and Appendix E include application case studies that describe how this is done in practice.

Stakeholders Participation Scheme

Often the stakeholders' viewpoints and preferences vary. For example, the end-users are mainly interested in using software, its performance, and the consequences of using the imagined DSS to their work. Normally the end-users evaluate software without knowing its internal details, or how it is developed. On the other hand, the developers
are responsible for producing software, which should satisfy quality requirements specified in the contract. They are therefore responsible for quality of the abstract prototypes as well as for quality of the final product.

Managers are typically interested in the overall quality rather than in the specific quality characteristics. Business related requirements are generally of more importance to them. They often strive after balancing quality improvement with management criteria such as schedule delay and cost overrun, and focus on optimization of quality within limited cost, human resources and time frame. Figure 4.5 shows a proposed general scheme for handling the preferences of the users, the developers and the managers in AP. It is basically a triad of consultations among the developers, the users, and the managers. The manager bears the ultimate responsibility in the DSS project, and must have the final say on the acceptability, apparently by making trade-offs of the demands of the customers (users), technological limitations, and business requirements.

Since various stakeholders have different preferences and requirements, an exclusive involvement of the users or the developers in the DSS design process would surely not be enough. The users' and the developers' collaboration scheme [refer to Figure 3.5] guides the participation of the developers and the future users in AP as the review subjects. It should be noted that in order to gain realistic and meaningful trends of opinions, it is important to involve as many appropriate subjects as possible.

4.1.3 Opinions Gathering

AP can be conducted at a single locality or in a geographically distributed environment. The latter form of AP is particularly well suited in the development of widely used DSS, as their stakeholders are typically distributed in geographically distributed locations. Two types of information are targeted, namely (i) the quantitative information, and (ii) the individual comments or suggestions. The former type of information can be defined, presented and processed algorithmically. The later kind of information cannot be processed automatically and decisions made based on this type of information depend entirely upon the skills of the developers. The developers assume the responsibility of facilitating the opinions gathering process, but must let the subjects give their views freely, without any external influence.
Figure 4.6 outlines the opinions gathering and processing procedure. Prior to opinions gathering, metrics to use in measuring the adequacy of the abstract prototype under consideration, and the kinds and number of subjects to engage in the review process must be agreed upon. An information-gathering matrix [see Figure D.1] and a representation of the abstract prototype in question must also be prepared. Further procedural guidelines on creation of the information-gathering matrix, and on opinions gathering are in Appendix D.

![Figure 4.6: Process of gathering opinions](image)

### 4.1.4 Application Scenarios

Figure 4.7 shows possible kinds of interplays between the requirements and realization of the expected DSS product. The main path to achieve the expected functionality leads through the theories, methods, algorithms, and pilot prototypes. This path is denoted by the acronym TMAPE. Having known suitable theories, the developers transform them to methods, then to algorithms, and finally to codes for pilot prototypes or the expected functionality. The TMAPE path guides the DSS design process through all basic intermediate milestones. There can also be other possible transformation scenarios, for instance, starting straight away to write codes for the expected functionality.

![Figure 4.7: General scheme for application of the AP technique](image)
based on theories (TE), theories and methods (TME), or theories, methods, and algorithms (TMAE). Furthermore, there are short cuts such as, writing codes for the expected functionality exclusively based on methods (ME), algorithms (AE), theories, algorithms and pilot prototypes (TAPE), theories and pilot prototypes (TPE), and methods and pilot prototypes (MPE).

In the development of a novel functionality, it can, however, be detrimental to jump directly into the development of methods, algorithms or pilot prototypes based on the understanding of the requirements (or needs) only, without finding the most appropriate theory or set of theories. Such practice can increase rework cycles. The reason is that such a DSS design process tends to be rather ad hoc (i.e. undisciplined and of trial and error in nature) and the end result would likely be prone to faults. On the other hand, the designs resulting from the TMAPE track are likely to be less vulnerable to errors since when gaps in the abstract prototypes are identified, information is fed back into the respective requirements specification model, and faults adjusted early on. In this way the rework cycles can be reduced or even avoided altogether. It can be said that in order to implement high quality DSS or its functionality from scratch, it is advisable to wholly follow the TMAPE path.

In some practical cases, standard methods, algorithms or codes that can approximately solve problems are available, and can be adapted and reused. For instance, if there is an implementation that roughly matches the expected functionality, a backward-forward path, that is, PAMT-TMAPE can be followed to scale the existing codes to the problem. Similarly, if there is an algorithm that closely matches the specifications for a problem, a backward path, for this case AMT, can be traced, followed by a forward transformation, that is, TMAPE, and so on. The challenge that the developers may face in reusing readily available abstract prototypes is how to interface them in an effective way, and how to synchronize the needed modification to the existing implementations. One of the dangers of following the reuse paths is that even mediocre concepts can unintentionally be institutionalized.

4.2 Computer Based Activities

One of the problems associated with AP is that some activities generate or require large amounts of information. Processing of such information manually would consume substantial amount of time, and the generated information must be stored and reused. Also, analysis of information involves immense routine calculations and the results need to be presented appropriately to facilitate interpretation. Thus, computer support of AP is deemed to be necessary. However, no ready-made computer based methods or algorithms that match the underlying demands are available. Therefore, there is a need to define formal procedural descriptions of the AP concept, with the eventual aim of characterization of the relevant activities into forms interpretable by computers. Computerization of the AP process is essential, and on the other hand, human participation in AP is also valuable and necessary. Requirements processing and information management
and analysis are activities that can partly be supported by computers. Other AP activities mentioned in Section 4.1.1 must be accomplished manually.

4.2.1 Requirements Representation and Processing

This section first describes a scheme for representation and clustering of requirements. It then presents a quasi-computerized methodology for preparation of the requirements models and review criteria for AP.

Representation and Clustering of Requirements

As previously stated, the requirements are formulated in advance in the analysis phase and used during the development interval. Requirements acquired in the previous projects can also be reused. To use requirements in AP, the developers must first cluster them according to the abstraction levels. However, some of the requirements can be valid in more than one abstraction level. Figure 4.8 shows a scheme for clustering requirements according to the abstraction levels and quality characteristics. It guides grouping of requirements, first depending on quality characteristics, and then according to abstraction levels. It also shows typical information items that must be specified alongside the requirement statements. These information items provide connectivity information and therefore play a central role when searching for the requirements. Ac-

![Diagram of Abstraction Levels, Quality Characteristics, and Requirements](image)

**Figure 4.8: Requirements representation and clustering scheme**

According to this scheme, a requirement is represented as a simple statement, with information items such as abstraction level, the functionality it relates to, the source of requirement, the stakeholder it concerns, the standard it belongs to, the characteristic it descends from, its applicability domain, and so on linked to it. A requirement statement and its related information items can be stored electronically in a searchable database. The advantage here is that when the developers look for requirements for a DSS, having clues about the problem, they can search and retrieve requirements that roughly match
the problem or project in question. The obtained requirement statements can eventually be fine-tuned to precisely match the problem or project.

In finding requirements from the database, the search mechanism can be, for instance, in Boolean query formulation [see Section 5.4.1] or in the form of 'IF THEN' statement (expressed in a mixture of natural language and computer language) as follows:

\[
\text{IF (functionality)\& (abstraction level) THEN (relevant characteristics/requirements/review criteria)}
\]

Thus, for an envisaged DSS feature at a given stage of the design process, a provisional list of requirements can be retrieved automatically after specifying appropriate conditions. In reality, quality requirements for complex DSS projects are enormous, and there is often the possibility to overlook key requirements. This approach can therefore be quite helpful in such situations. A large number of requirements can be acquired, represented, and stored in advance, and unveiled to the developers when needed. This helps ensure completeness of the requirement models.

The process of formulation of requirements for the elementary functions\(^4\) presents a tree-structure, where the leaves are the requirements related to various different quality characteristics. Meanwhile, the requirements for components of the envisaged DSS such as input, output, user interfaces and database give rise to requirements for the overall software system. There are also the so-called assembly requirements, which can be described as requirements associated with clusters of functions that depend on or operate on each other.

**Preparation of Requirements Model and Review Criteria**

The starting point in the process of preparing the requirements model for a DSS in the design stage is the database, in which all kinds of requirements collected in the requirements specification phase are stored. One of the inputs to this process is the identity of the abstraction level. Furthermore, since a function of a DSS can stand for one or several needs, the identity of the function can also be used as a keyword for searching requirements. Computerization of the searching procedure enables automatic generation of a provisional requirements model, which contains requirements that can eventually be fine-tuned and ultimately converted to the review criteria.

The procedure can formally be described as follows. For a given function \(f_j\) at a given abstraction level \(l\), sets of generic requirements \(R_u = \{g_1, g_2, \ldots, g_m\}\), \(R_u \subset R\) and specific requirements \(R_s = \{s_1, s_2, \ldots, s_n\}\), \(R_s \subset R\) are retrieved from a pool of available requirements \(R\), where \(R = \{g_1, g_2, \ldots, g_p, s_1, s_2, \ldots, s_q\}\). The generic and specific requirements \((R_u \cup R_s)\) are then adapted to the project. Requirements considered to be irrelevant are left out and a set of fine-tuned requirements \(R_f \subset (R_u \cup R_s)\) is generated. Purging an extensive list of requirements to a short, manageable list of the most relevant ones is

---

\(^4\) In the context of this work, the hierarchical decomposition of DSS system includes the software system at the highest level, modules, functions, sub functions, and at the lowest level the elementary functions, which practically equal to the menu commands.
typically based on the expert judgment of the developers. New additional requirements $R_n$; where $R_n \cap R = \emptyset$ may also be formulated. Consequently, a requirements model ($R_f$), typically contains fine tuned requirements ($R_t$) and new requirements ($R_n$), where $R_f = R_t \cup R_n$. Figure 4.9 summarizes this procedure.

One of the strengths of this semi automatic procedure is that it mitigates the effects of it being purely automatic. It allows the DSS developers to participate or intervene, and helps reduce the danger of overlooking requirements. It can also fasten the requirements acquisition process [see case studies in Chapter 6].

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{diagram.png}
\caption{Formulation of requirements and review criteria}
\end{figure}

4.2.2 Knowledge Acquisition, Representation and Processing

As mentioned in Chapter 2, knowledge builds up as the DSS development progresses. Figure 4.10 shows the sources of knowledge, and how information accumulates during
abstract prototyping. There are four kinds of information in this context, namely (i) the requirements, which describes how the envisaged abstract prototype will be like, (ii) information about possible solutions (i.e. alternative solutions) to the problem in hand, (iii) field information, which is mainly made up of 'raw' data from the subjects, and (iv) the analysis and interpretation results. These kinds of information as well as guidelines for AP can be stored and reused. Evidently, there is a challenge of handling the information generated during AP. The following sections describe how the generated knowledge can be appropriately modeled, represented, and eventually processed for use in AP.

**Representation of Solution Concepts**

There are several methods and tools used in software engineering and other fields to represent concepts, or to support discussions or clarification of concepts. These include, for instance, texts, scripts and scenarios, paper prototypes, storyboards, on-line prototypes, throwaway rapid software prototypes, evolutionary rapid prototypes, video recordings, photo prints, sketches, virtual reality models, and reasoning models [Bayer and Holtzblatt (1998); Casaday and Cynthia (1995); Owen and Horváth, (2002); Uden (1995)]. These methods are primarily used to describe solution concepts during software conceptualization and design. Some of them can be adopted and applied in AP as well, and used as means to represent concepts or for communicating ideas among the developers or between the developers and the subjects or the management.

Figure 4.11 shows the proposed methods of representation of abstract prototypes. Practically, any of the proposed methods can be used, but the selection depends on, among other things, the effectiveness of the method. In the selection of the representation method, each of the alternative methods must be evaluated relative to a set of screening criteria. Only methods that virtually fulfill the screening criteria can advance to the candidates' list. "Yes" or "No" answers can be used to seed methods and quickly filter out the methods that fail to meet key selection criteria. This helps narrow down the list of methods of representation to a shorter list that receives serious consideration.

---

5 The terms 'knowledge', 'information', and 'data' have been defined in Section 3.7.
The goal is to enable subjects to be sufficiently informed, interested and involved in AP by providing them with models or representations that are interesting and that effectively serve as means for clarifying concepts. Sometimes different forms of representation can be used simultaneously to help subjects better understand the abstract prototypes. Existing software tools such as text editors, equation editors, 2D modelers, and 3D modelers can be customized and used in the preparation of the representations of the abstract prototypes. In some instances, a template can be prepared in advance such that once a concept is thought of, it can be represented based on the template. In this case, only specific information that describes the abstract prototype in question needs to be added. This provides an effective way for representing the abstract prototypes quickly and in a standardized way.

**Tracking Related Abstract Prototypes**

A procedure for tracking the abstract prototypes related to the developed or chosen abstract prototype of the DSS at the design phase is proposed. The goal is to know if the developed or chosen abstract prototype fits, and can be integrated and used without any adverse consequences alongside other abstract prototypes. Let \( t \) be a theory, where \( t \in \mathcal{T} \)
and $T$ is a set of theories for a given DSS; $m$ be a method, where $m \in M$ and $M$ is a set of methods underlying the DSS; $a$ be an algorithm, where $a \in A$ and $A$ is a set of algorithms for the DSS; and $i$ be a pilot prototype, where $i \in P$ and $P$ is a set of pilot prototypes that make up the DSS. Let also $N = \{n_1, n_2, \ldots, n_k\}$ be a set of key requirements (i.e. the requirements that must be met unconditionally). The process of tracking the abstract prototypes related to $t$, $m$, $a$, and $i$ can be generalized as follows.

$$
\kappa : \{t \rightarrow \Lambda^t \\
m \rightarrow \Lambda^m \\
a \rightarrow \Lambda^a \\
i \rightarrow \Lambda^i\}
$$

where; $\kappa$ is the tracking mechanism and $\Lambda^t \subseteq (T,M,A,P)$, $\Lambda^m \subseteq (T,M,A,P)$, $\Lambda^a \subseteq (T,M,A,P)$, and $\Lambda^i \subseteq (T,M,A,P)$ are sets of the abstract prototypes related to $t$, $m$, $a$, and $i$ respectively. $\kappa$ vary with $N$. Specifying the elements of $N$ and applying it enables the identification of the related theories, methods, algorithms, and pilot prototypes. It is thus possible to explore the implications of the choices made, and to know in advance if the developed or chosen abstract prototype conforms to other solution elements. Also, this procedure in a way matches DSS projects to the abstract prototypes. If the needs (and therefore the requirements) are known, then the related abstract prototypes can be sought and subsequently adapted and reused. The developers can thus simply reengineer the existing abstract prototypes to satisfy specific requirements. The advantage here is that if there are similar abstract prototypes, the DSS development process can be reduced to recalibration of the existing solution, to incorporate only specific missing features.

### 4.2.3 Information Analysis

Information analysis in the AP context includes: (i) processing of the data that represent views of the representatives of the DSS stakeholders on quality of the abstract prototypes, and (ii) interpretation of the results. This includes computation of quantitative empirical indicators, to quantify the acceptability of the abstract prototype in question and the extent to which it satisfies its requirements. In the processes of conceptualization and designing DSS, identification of weak spots and/or selection of solution proposals are among the key activities. Often single or multiple solutions come into the picture and reviews and/or selections need to be made objectively, by taking into account opinions of various stakeholders. The evaluation scheme [Figure 4.12] provides quantitative means for achieving this. It brings forward a framework for: (a) arranging alternative solutions in the pecking order, and for (b) determination of the extents of fulfillment of the requirements. Arrangement of the alternative solutions in the pecking order is based on an index called acceptability index ($\alpha$).

Subjects are required to specify the level of fulfillment of each requirement ($\phi$), relevance of every requirement ($p$), and their confidence ($\chi$). The specified values depict
how the subjects feel the requirements have been fulfilled, are relevant, and how accurate they responded respectively. For \( m \) review criteria used in an AP review in which \( n \) subjects participated, the total merit value \( \mu_r \) for a solution proposal can be determined as follows:

\[
\mu_r = \sigma_{ij} \sum_{i=1}^{m} \sum_{j=1}^{n} \phi_{ij} \rho_{ij} \chi_{ij} \quad \phi_{ij} \geq 0 \\
\quad \quad \rho_{ij} \geq 0 \\
\quad \quad \chi_{ij} \geq 0
\] (4.2)

where \( \phi_{ij} \), \( \rho_{ij} \), and \( \chi_{ij} \) are the level of fulfillment, the relevance, and the confidence about requirement \( i \) as expressed by subject \( j \) and \( \omega_i \) is the weight assigned by the developer (the requirements engineer), which signifies the importance of the requirement \( i \). The acceptability index (\( \alpha \)) is then defined as follows:

\[
\alpha = \frac{\mu_r}{\epsilon_s}
\] (4.3)

where, \( \epsilon_s = n m (\phi_{\max} \rho_{\max} \chi_{\max}) \). \( \epsilon_s \) is known as the maximum possible total merit value. \( \phi_{\max} \), \( \rho_{\max} \), and \( \chi_{\max} \) are the maximum achievable values of \( \phi \), \( \rho \) and \( \chi \) respectively. \( \alpha \) gives the developers clue on the extent to which the proposed solution matches the ideal solution. Based on the values of \( \alpha \), the solution proposals can be ranked in the order of salience. \( \mu_r \) values for the requirements can also be presented graphically i.e. used in plotting the so called \( \sigma - \) diagram [Figure D.5]. On the other hand, collected \( \phi \) values can be used in the generation of the \( \phi - \) diagram [Figure D.6]. These diagrams provide visual overview of the extents of fulfillment of the requirements. The \( \rho \) and \( \chi \) values are also used in the determination of the \textit{relevance} (\( P_i \)) and \textit{confidence} (\( X \)) indices respectively as follows.

\[
P_i = \frac{\sum_{j=1}^{n} \rho_{ij}}{n \rho_{\max}}
\] (4.4)

\[
X = \frac{\sum_{i=1}^{m} \sum_{j=1}^{n} \chi_{ij}}{mn \chi_{\max}}
\] (4.5)

The index \( P_i \) indicates how relevant, in the opinion of the subjects, the review criterion and consequently the requirement is, with regard to the ongoing AP review. The index \( X \) indicates how assertively the subjects have assigned \( \phi \) and \( \rho \) values, and gives clue on how knowledgeable and reliable they have been. Appendix D provides further elaboration of the proposed information analysis process.

It is; however, important to emphasize that what has been introduced in this section is just an empirical procedure for gathering and analysis of opinions designed specifically for the purposes of this research. In practice, the developers can choose to adopt this
procedure or create their own opinions gathering and analysis methods. They may even
device their own procedures for measuring the acceptability of the abstract prototypes,
and apply them in the framework of AP.

4.3 Conclusion

This chapter described the AP process and presented formal methods for shaping and
guiding various manual and automated AP activities. It also presented the methods
upon which software tools that support the AP activities can be based. It has been
shown that the AP process is epitomized by systematic and recursive prototyping, re-
view, selection, and improvement of the abstract prototypes. AP has in the first place
been developed as a procedure for pre-implementation testing of any DSS. It offers a
structured framework, which the DSS developers can apply to shape and direct activi-
ties in the design phase. It also lays down a procedure that can be used to acquire and
continuously take on board requirements and to monitor their fulfillment as the process
of designing DSS progresses. Another unique feature of AP is that it provides schemes
for handling knowledge about the abstract prototypes and for involving the representa-
tives of various DSS stakeholders in assessing the consistence, completeness and us-
ability of the abstract prototypes. The goal is to ensure that the eventual DSS meets the
demands of the users as well as the constraints posed by the state of the art technolo-
gies. Such stakeholder oriented pre-implementation testing increases the chance of cre-
ating most appropriate DSS. AP can be incorporated in any modern software develop-
ment framework as a design model, or as an embedded prototyping and review proce-
dure for checking, at various abstraction levels, whether the requirements are being ful-
filled. Some of the AP activities can be done exclusively by human beings while others
can be supported by computers. The following chapter addresses the issue of comput-
erization of the AP process.
Chapter 5

Design and Implementation of the Abstract Prototyping Software

This chapter addresses issues related to design and realization of a pilot implementation of the AP software. It first specifies the computer supported activities and describes how modules and features were allocated to the AP software. It then presents the requirements and the design of the AP software, and explains how the functions of the AP software were implemented.

5.1 Needs Analysis and Allocation of Features to the Abstract Prototyping Software

As mentioned earlier, some of the AP activities can be automated fully or to some extent. It was thus important to determine to what extent the AP process could be computerized, and to allocate modules and features to the AP software. The activities supported by computers include requirements processing, preparation of the representations of the abstract prototypes, preparation of the opinions gathering matrices, and information analysis.

Table 5.1 shows the roles - module matrix, which served as the basis for allocation of features to the modules of the AP software. In AP, an individual can play the role of (i) a subject, (ii) a developer, (iii) an author of the representation of an abstract prototype, (iv) a requirements manager, (v) a process controller, and (vi) an information administrator. The subjects co-determine the suitability of the abstract prototypes and give their judgments via the information gathering matrices in local or geographically distributed environments. As previously mentioned, their views are subsequently analyzed and the best proposals selected, flaws detected, and an improvement strategy established. Hence, subjects need to be equipped with on-line software tools for browsing and visualization of the representations of the abstract prototypes and for communicating their responses electronically, for instance, via email or web. The developers are responsible for preparation and running of the AP reviews. This includes selection of the review subjects and authoring of the representations of the abstract prototypes. They ultimately analyze and interpret the results. Hence, they need automated tools for supporting the
preparation activities and for supporting the analysis of field information. These include software tools for supporting acquisition of requirements for the abstract prototypes, computation of $\alpha$, $X$, $\Phi$ and $P$ indices, and for supporting generation of the relevant diagrams.

### Table 5.1: Activities supported by the AP software

<table>
<thead>
<tr>
<th>Stakeholder / Module</th>
<th>Requirements representation and processing</th>
<th>Knowledge representation and processing</th>
<th>Opinions gathering</th>
<th>Information analysis</th>
<th>Data and knowledge management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developer</td>
<td>Input/delete/edit/browse requirements</td>
<td>Browse knowledge on abstract prototypes</td>
<td>Prepare tests Receive, Send</td>
<td>Input/delete/edit/browse data, represent results</td>
<td>Browse, match problem to requirements</td>
</tr>
<tr>
<td>Information manager</td>
<td>Annotate</td>
<td>Annotate</td>
<td>Annotate</td>
<td>Annotate</td>
<td>Input/delete/edit/browse data</td>
</tr>
<tr>
<td>Process controller</td>
<td>Annotate</td>
<td>Annotate</td>
<td>Annotate</td>
<td>Annotate</td>
<td>Input/delete/edit/browse, annotate</td>
</tr>
<tr>
<td>Authors of the representations</td>
<td>Browse requirements</td>
<td>Input/delete/edit/browse knowledge</td>
<td>-----</td>
<td>-----</td>
<td>Browse abstract prototypes</td>
</tr>
<tr>
<td>Requirements manager</td>
<td>Input/delete/edit requirements</td>
<td>Browse knowledge</td>
<td>-----</td>
<td>-----</td>
<td>Browse requirements</td>
</tr>
<tr>
<td>Subject</td>
<td>Suggest additional requirements</td>
<td>Browse knowledge on abstract prototypes</td>
<td>Input/delete/edit/submit values</td>
<td>Obtain feedback</td>
<td>Browse data</td>
</tr>
</tbody>
</table>

During the analysis, the developers should also be able to: (i) visualize the representations of the abstract prototypes, and (ii) explore relationships among theories, methods, algorithms and pilot prototypes. Consequently, they also need automated means to support these activities as well. The authors of the representations are responsible for creation of the representations of the abstract prototypes. Hence, they should be offered with software tools for supporting them in building, inputting, editing, browsing and storing the representations. The requirements managers are responsible for the acquisition and management of requirements. Thus, they need automated means to support input, editing, storage, and browsing of requirements. The process controllers oversee whether the AP process proceeds, as it is supposed to, while the information administrators are responsible for the integrity of the warehouse\(^1\). Hence, they both need to have full access to information and to control the information system. Software tools supporting processing of the requirements information, field data, current and previous analysis results, information about the abstract prototypes, and manipulation of information in the database should be availed to them.

### 5.2 Requirements for the Abstract Prototyping Software

The AP software must effectively address the demands stipulated in Section 5.1. The requirements for the AP software can therefore be summarized as follows. It should support or help:

\(^1\) Warehouse is a general term used in this work to refer to repository of data or knowledge, that is, a database or a knowledgebase, including data or knowledge handling tools.
• Management of requirements. In this regard, it should be noted that:

◊ The developers are responsible for the acquisition of the requirements, which are the sources of the review criteria [refer to Chapters 3 and 4]. Requirements for various functions should be represented as simple statements, grouped according to the abstraction levels, and stored. Utilities to support searching, input, editing, storing and visualization of requirements must be provided.

◊ The information items that accompany a requirement statement include the identity of the function it relates to and of the abstraction level, and quality characteristic it belongs to. Others are the source of the requirement, whether the requirement is specific or generic, whether the requirement is functional or non-functional, and the identity of the stakeholder it relates to. These information items should be pre-defined to facilitate data entry.

• Authoring of the representations of the abstract prototypes and availing them in various forms of representations. The following should be noted in this respect.

◊ The developers are required to (i) select the method of representation; (ii) prepare the representation; and (iii) input the representations of the abstract prototypes into the database. Thus, software tools to support them in making, inputting, editing and storing the representations of the abstract prototypes are necessary. Also required is an effective access to information about the abstract prototypes, at any geographical location.

◊ Several kinds of information can accompany a representation of an abstract prototype. Such information items include the name of the abstract prototype, the identity of the function, the related abstract prototypes, and the abstraction level it belongs to. Information for administrative purposes such as the author's name and the preparation date can also be incorporated. These information items should be pre-defined to facilitate input of data.

• Handling the opinions gathering activities. In this regard, software must:

◊ Support preparation of information-gathering matrices.

◊ Facilitate sending information-gathering matrices (together with the representations of the abstract prototypes) to the subjects in local or geographically distributed locations.

◊ Facilitate input of the requested information items, namely (i) the relevance of the review criteria to the problem in question, \( r \); (ii) the degrees of fulfillment of the review criteria by the abstract prototype in question, \( \phi \); and (iii) their confidences with regard to values they allocate, \( \chi \).

◊ Provide means for sending back the completed matrices.

• Analysis of information. Having generated alternative solutions, the developers are often faced with the problem of selecting the most appropriate ones. This must be
done precisely and objectively. Furthermore, there is a need for the values involved in the determination of the levels of fulfillment of requirements to be accurate and reliable. As previously mentioned, based on these values, the requirements that have not been fulfilled adequately can be singled out, and deficiencies in theories, methods, algorithms or pilot prototypes revealed. Hence, computers must support the developers to perform the relevant computations, having the necessary inputs (i.e. $\phi$, $\rho$, and $\chi$) from the subjects. However, the developers must fully control the analysis process. The information analysis software should therefore support:

- Determination of the acceptability of the abstract prototypes.
- Determination of the level of fulfillment of the requirements.
- Ranking of the requirements.
- Establishment of what to improve.

Software should also support visualization of the representations of the abstract prototypes and exploration of their relationships to other adopted solutions. It must also support the activities performed prior to opinions gathering, e.g., a matching problem to the available requirements.

- Storage of requirements, representations of the abstract prototypes, and analysis results. In this regard:
  - Tools to facilitate input, editing and updates of information should be incorporated.
  - Access to information from the functional modules of the AP software should be provided.

Computer support of AP is important in many respects. For example, computers enable massive amounts of information about the abstract prototypes to be stored and in turn processed and availed to the subjects or the developers even at distant geographic locations, allow for unlimited number of requirements to be stored, and enable the computations required during the information analysis stage to be performed faster and accurately.

5.3 Design of the Abstract Prototyping Software

Software design is concerned with the engineering of the structure and quality of software [see e.g. Fairley, 1985]. As indicated in Sections 5.1 and 5.2, the AP software is expected to offer numerous tools to support various AP activities. This section presents the structural and functional design of the AP software.
5.3.1 Architectural Design

The AP software is organized in a modular fashion and integrated into one system [Figure 5.1]. The functional modules include those for requirements representation and processing, knowledge representation and processing, opinions gathering, and information analysis. Each module may consist of the main functions, sub-functions of various levels, and elementary functions. An elementary function practically equals to the menu commands. A warehouse (i.e. database and knowledgebase) and several system management and auxiliary utilities have also been incorporated. The user interface provides links to the functional modules and keeps information available whenever requested.

The main goal of the requirements representation and processing (RRP) module [Figure 5.2] is to support acquisition of requirements. It provides software tools, which support input, editing, storage, visualization, and processing information about the requirements. The strategy is to accumulate various kinds of requirements and to provide the developers with sorted requirements.

The main function of the knowledge representation and processing (KRP) utility [Figure 5.3] is to support management of knowledge and to provide the developers and the subjects involved in AP with effective access to information about abstract prototypes, regardless of their geographical locations. It offers tools for storage and processing of information about the abstract prototypes. Knowledge representation and processing is well suited to computer assistance because computer can effectively handle large amounts of information and avail
the information in various different forms and formats. The motivation behind building the KRP utility is to enable and to facilitate reuse of previous theories, methods, algorithms or pilot prototypes. The goal is to enrich the developers as well as the subjects with information about these abstract prototypes. The idea is to use the representations of the abstract prototypes as communication means when judging if the proposed solutions are suitable. The KRP module also accommodates software tools, which support the developers to retrieve the abstract prototypes or the information related to them based on one or more search criteria.

The opinions gathering module [Figure 5.4] supports the developers in the preparation of the information gathering matrices and in sending the electronic review materials, namely the information gathering matrix and the representation of the abstract prototypes to the subjects. It also comprises software tools, which supports the subjects in sending back the completed information-gathering matrices.

The information analysis module [Figure 5.5] supports analysis of opinions (i.e. field data). It also supports presentation of the analysis results. Furthermore, it comprises algorithms that can be applied in the determination of priorities to give to various requirements, and in ranking the requirements according to their relevance.
In addition, it equips the developers with visualization tools (i.e. for visualizing the representations of the abstract prototypes) and with tools that can be used in exploring the relationships and the consequences of the adopted solutions.

![Diagram of GUI of the Abstract Prototyping Software](image)

**Figure 5.5: Structure of the information analysis utility**

### 5.3.2 Information Flow

The concept of Petri-nets [Fairley (1985); Molloy and Peterson (2000); Murata (1989); Staufer (1987); van der Aalst (1998)] has been used in various problems to portray information flow. It is adopted in this work to describe the information flow in AP, taking the advantage of its capability and adequacy of modeling processes. One of the main benefits of using Petri-nets in modeling processes is that the states in the process can be explicitly and adequately described.

Theoretically, AP is a structured procedure, formed by sets of changes of the states $s_i$. The individual states are described by the state parameters $p_{ij}$. The set of state parameters $S$ is a union of parameters $p_{ij}$ related to the execution of the AP process as well as to the global DSS development. A state, $s_i$, of the AP process is said to change when at least one parameter, $p_{ij}$ is changing, and different states occur when time is available for an activity to be carried out, and for a change to take place. An AP process is built up from a finite number of states $s_i$, and the states can be distinguished based on the changes. Clearly there is an analogy with the concept of Petri-nets as follows. The states of the AP process can be compared to places, $p_{ij}$, of the Petri-net. The changes between the states of an AP process can be compared with transitions, $t_{ik}$, of the Petri-net.

In order to use Petri-nets for modeling the AP process, the following have been done: (i) identification of the states of the AP process; and (ii) specification of the transitions. Figure 5.6 shows the flow of information in the AP process, and specifies the relationships among various high-level functional elements. Only high-level functional elements are dealt with because the building blocks of the AP software are mostly units or utilities of the existing software tools that have been tested and used successfully else-
where. This means that the embedded low-level functions are inherited and reused. The nodes (that is, the places and the transitions) specify the process activities. Description of the logical relationships can be extended to all levels of functional decomposition and cannot for sure be neglected at the elementary functions level, for the reason that the input/output related to a particular function can only be defined at this level.

Figure 5.6 not only represents the flow of information, but also presents the sequence and the dependencies of various activities supported by the AP software. Based on this diagram, highly modular software, which allows for the reuse, extension or removal of parts of the software at minimum effort and costs has been created.

Figure 5.6: Information flow in AP

5.4 Implementation

The component reuse software development model\(^2\) [Jones, 1990] was adopted and used as a concept model for shaping and directing activities during the development of the AP software. The main reason of choosing the components reuse model was economy. It provided an economical way of creating new software in that the implementation process largely involved identification and integration of the existing codes (that have previously been tested and used in other areas and for other purposes), or fully-fledged functions, utilities or applications, sometimes without any sizeable modifica-

\(^2\) Refer to Section 2.3.1 for further details on the component reuse model.
tion; and using an interface as glue. The main challenge was how to interface components in an effective way in order to satisfy the requirements for the AP software, and how to synchronize the needed modification to the existing codes. The strategy was to utilize the components of the widely used commercial software products as much as possible in order to speed up the development. This also ensured portability and compatibility. Large portion of the AP software has been built using Microsoft technology, and the source code has been written mainly in Visual Basic, Visual Basic for Applications, Visual C++ and HTML. There was no significant advantage from research point of view of developing everything from scratch since similar utilities were readily available in other existing software packages, and could be used to sufficiently demonstrate how computer can support the AP process. This section explains how various software tools for supporting various AP activities have been implemented.

5.4.1 Requirements Representation and Processing Module

The requirements representation and processing (RRP) module embraces software tools, which support finding requirements by specifying certain search criterion or set of criteria, updating the requirements database, and browsing requirements in the database.

Multi Dimensional Requirements Search

A structured query language (SQL)\(^3\) based relational database management system (RDBMS) is used in the AP software to support data and knowledge storage. It provides data links that allow the developers to access tables and various information items as needed. Figure 5.7 shows Boolean query formulations for the implemented multi-dimensional requirements search (MDRS) utility. The MDRS algorithms have been developed to assist the developers in finding various kinds of requirements models for different combinations of search criteria. Basically this involves usage of more than one keyword in seeking the relevant requirements. The implemented MDRS queries include those for finding: (a) all requirements corresponding to a given task, (b) requirements that relate to various quality characteristics, (c) specific or generic requirements, (d) requirements of a known stakeholder, and (e) requirements for a functionality. The pseudocodes below show how these functions have been implemented in SQL.

<table>
<thead>
<tr>
<th>Syntax for searching requirements in the database</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SELECT</strong> Characteristic, Requirement Statement.</td>
</tr>
<tr>
<td><strong>FROM</strong> [Requirements Database]</td>
</tr>
<tr>
<td><strong>WHERE</strong> (Conditions expressed in Boolean Formulation i.e. AND, OR, NOT);</td>
</tr>
</tbody>
</table>

---

\(^3\) Structured Query Language (SQL) is a programming language developed specifically to access data arranged in tables of rows and columns, as in a database, as well as to do searching, sorting and cross-referencing of data. Refer to, e.g. http://rute.sourceforge.net/.
The MDRS utility provides sorted requirements, which must, however, be put in the context of the current DSS project. In other words, requirements derived from the database are in most cases incomplete and need to be finalized by the developers. This can be accomplished, for instance, by holding a brainstorming session in which the computer generated provisional list of requirements is fine-tuned and made specific to the task in hand. Such human participation is worthy, and is needed in requirements processing. The dynamic character\(^4\) of requirements must also be taken into the consideration. New requirements frequently come into the picture in the development interval, and should be considered and added as appropriate.

\[
\left\{ \text{Abstraction level} \right\} \text{ and } \left\{ \text{Functionality} \right\} \text{ and } \left\{ \text{Applicability or generic} \right\}
\]

(a) Finding the requirement model for a task

\[
\left\{ \text{Abstraction level} \right\} \text{ and } \left\{ \text{Characteristic} \right\}
\]

(b) Finding requirements for a characteristic

\[
\left\{ \text{Abstraction level} \right\} \text{ and } \left\{ \text{Functionality} \right\} \text{ and } \left\{ \text{Applicability} \right\}
\]

(c) Finding specific or generic requirements

\[
\left\{ \text{Stakeholder} \right\} \text{ and } \left\{ \text{Functionality or Abstraction level} \right\}
\]

(d) Finding requirements for a stakeholder

\[
\left\{ \text{Abstraction level} \right\} \text{ and } \left\{ \text{Functionality} \right\}
\]

(e) Finding requirements for a functionality

**Figure 5.7: Boolean query formulation for the MDRS utility**

**Updating the Requirements Database**

A form is provided through which the user can update (i.e. modify, add, or delete) requirements in the database without fully running the basic database software. ActiveX\(^5\) controls display data from the database records (i.e. the requirements statement, quality

\(^4\) 'Dynamic character' refers to endless emergence of requirements in the development interval. Also, it is a common phenomenon in requirements engineering that as needs and technologies change, new requirements always evolve. However, it is difficult for the developers to ensure that all desirable characteristics and requirements have been taken into the account, as there are always new needs and new technologies at the marketplace.

\(^5\) ActiveX can be described as set of technologies based on Microsoft's Component Object Model (COM) for creating reusable binary objects [refer to http://microsoft.com/]}
characteristics it relates to, its source, and so forth) on a form. This form is linked to the GUI of the AP software by the function below placed in the 'main' part of the source codes:

```
Function Link_Update_Interface()
  Load Updates Form
End of Function
```

The requirements update interface [Figure A.2] is implemented such that pre-defined values are provided to facilitate entry of the information items that accompany requirement statements.

**Browsing Requirements**

Means by which the users can visualize the requirements in the database without opening it are provided. ActiveX controls display data from database records (i.e. the requirements statement, quality characteristics it relates to, its source, and so on) on the 'requirements browser' form. This form is linked to the GUI of the AP software by the following function.

```
Function Link_Requirements_Browser_Interface()
  Load Requirements Browser Form
End of Function
```

### 5.4.2 Knowledge Representation and Processing Module

As an attempt to deal with the challenges of handling knowledge during AP, a computer-based knowledge representation and processing (KRP) utility has been implemented. This utility consists of software tools that support representation and update of information about the abstract prototypes, and it also supports information search.

**Authoring the Representations of Abstract Prototypes**

The KRP utility provides software tools that support authoring of the representations of the abstract prototypes. These tools, as established in Chapter 4, include those for text, graphics, video and audio processing. The pseudocodes below show how such tools have been incorporated into the AP software.

```
Function Call_Knowledge_Representation_Tool()
  Load Form containing representation tool
End of Function
```

This is achieved simply by loading a user-prepared form. This is a specially designed user interface that consists of an executable physical file of the relevant authoring tool (commonly known as ActiveX component or formerly as OLE automation server) that contains classes from which objects can be defined. In this way, the Microsoft's Equa-
tions editor, Word, Visio, Drawing, and PowerPoint applications have been made available to support authoring of the representations of the abstract prototypes.

**Inputting the Representations of the Abstract Prototypes in Text, Audio, Exe, or Video Forms**

A form is provided through which the user can input the information about the abstract prototypes in textual, audio, exe, or video forms into the database. An OLE Object\(^6\) makes this possible. The information input form is linked to the GUI of the AP software as follows:

```plaintext
Function Link_Input_Interface()
    Load Input Form
End of Function
```

To input a representation in a database, a user either has to copy and paste a file onto the OLE object, or to cut the actual text or graphic object, and paste it onto the OLE object.

**Multi-Dimensional Knowledge Search**

The MDKS utility provides the developers with means for: (i) finding the relevant abstract prototypes for a problem, and (ii) for browsing the database. Some utilities found in the widely used commercial RDBMS have been incorporated in the AP software to demonstrate the MDKS concept. The pseudocodes below show how the search mechanism has been implemented in SQL.

```plaintext
SELECT (Required information),
FROM [Database]
WHERE (Conditions expressed in Boolean Formulation i.e. AND; OR; NOT);
```

In using the MDKS utility, the user is required to specify search dimensions\(^7\) for the intended exploration.

**Browsing the Representations**

Means by which the users can visualize the representations of the abstract prototypes in the database without opening it is provided. ActiveX controls display data from the database records (i.e. the actual representation, its identity, the abstraction level it corre-

---

\(^6\) Refer to [http://www.microsoft.com/](http://www.microsoft.com/)

\(^7\) A search dimension is a criterion (i.e. a keyword) or combinations of criteria (keywords) specified when searching for appropriate abstract prototypes or applicable requirements from the AP software's database. For example, in finding the existing abstract prototypes or the requirements for a problem at a given abstraction level; the search dimensions can be the identities of the 'functionality' and of the 'abstraction level'.
sponds to, and so forth) onto a form. The form is in turn incorporated as an integral part of the AP software as follows.

Adding the web browser form to the AP software

<table>
<thead>
<tr>
<th>Function Link_AbstractPrototypes_Browser_Interface()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Abstract Prototypes Browser Form</td>
</tr>
<tr>
<td>End of Function</td>
</tr>
</tbody>
</table>

5.4.3 Opinions Gathering Module

The AP software supports the developers in the preparation of the information-gathering matrices, and provides them with means for sending them (as well as the representations of the abstract prototypes) electronically to the subjects. A tool for preparing instructions is also provided.

Creation of an Information Gathering Matrix

Information gathering matrix is basically a paper-based document or an electronic file equipped with macros\(^8\). It is availed as a standard formatted document, with the review criteria for the task in hand inserted into it. The pseudocodes below show how this has been achieved.

Using Excel in creating information gathering matrix

<table>
<thead>
<tr>
<th>Function Create Information_Gathering_Matrix()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declarations</td>
</tr>
<tr>
<td>Set {the Excel sheet</td>
</tr>
<tr>
<td>The application visible &amp; active}</td>
</tr>
<tr>
<td>Set {Standard texts}</td>
</tr>
<tr>
<td>Load {requirement statements &amp; other relevant</td>
</tr>
<tr>
<td>information items }</td>
</tr>
<tr>
<td>Set format and colors in cells by running macros</td>
</tr>
<tr>
<td>End of Function</td>
</tr>
</tbody>
</table>

Sending Review Materials and Submission of Data

The information-gathering matrix reaches the subjects as a file with macros [Figure A.17]. These macros help automate some of the matrix completion activities. The completed form can in turn be sent back to the developers via email or submitted via the web. The web-based function below has been incorporated in the AP software to cater for these needs.

---

\(^8\) A macro is a series of commands that the basic application carries out automatically, in this particular case, to automate the matrix completion task.
Loading a web-based file upload function

Function_Load_Upload

Set Form size
Specify the destination URL address
Launch Web Browser
If busy Then
notify
Else
Indicate that it is "Working..."
End If
End of Function

Structure of the HTML file submittal codes

<HTML>
.
.
Codes for uploading personal/company details
Codes for uploading the file
File submittal codes
.
.
</HTML>

The information gathering matrices and the representations of the abstract prototypes can also be published and made available via the web. The Internet is an effective method for collaborating geographically distributed DSS stakeholders in AP. In the Internet set up, remote subjects can download the information-gathering matrices or access a server and deliver completed matrices [Figure A.15]. Similarly, they can browse and visualize the representations of the abstract prototypes published on the web, via the Internet. This sets up an effective platform for quick collection of opinions from a wide spectrum of stakeholders.

Preparation of Instructions for Opinion Gathering

The instructions on how to conduct an AP review are prepared using traditional word processors accessible via the AP software's GUI. The pseudocodes below show how a template document of instructions is called via the GUI. This is achieved simply by loading the template file using this function, and this automatically launches the word processor application.

Linking to .doc instructions file

Function_Call_Instructions_File()
.
.
Declarations
Set {the Word editor
The application visible & active}
Call template instruction sheet ("c:\ -.doc")
.
End of Function
5.4.4 Information Analysis Module

Several functions designed to support various information analysis activities have been implemented. These include those for determination of: (i) the levels of fulfillment of requirements (Φ) and presentation of the results graphically or in tabular form; (ii) the acceptability index (α); (iii) the relevance index (R); and (iv) for computation of the confidence index (X). As mentioned earlier, these indices are dependent upon the χ, φ and ρ values specified by the subjects. Based on these data, the parameters required in the determination of the above-mentioned indices as well as in the generation of the relevant graphs such as the averages and the standard deviations of scores, are computed. α and Φ indices give indications on how close the ideal solutions have been approached and on the extent to which the abstract prototypes fulfill requirements respectively, whereby R and X indices indicates how reliable the review process has been. Also, χ, φ and ρ values provide the basis for: (a) ranking the requirements, and (b) for deciding on the strategy for improvement of the abstract prototypes. Having known the needs, the relevant requirements can also be traced.

Algorithm for Determination of Levels of Fulfillment of Review Criteria

The χ, φ and ρ values are the inputs of the algorithm for determination of the level of fulfillment of the review criteria. The intermediate milestones in this process include computation of the total scores [i.e., \( \mu_j = \sum_{i=1}^{n} \phi_i \rho_i \chi_i \)] and the maximum possible scores [i.e., \( \varepsilon_j = n(\phi_{\text{max}} \rho_{\text{max}} \chi_{\text{max}}) \)]. The pseudocodes below describe the function that determines the levels of fulfillment of the review criteria.

<table>
<thead>
<tr>
<th>Determination of the levels of fulfillment of review criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Determine LevelsOfFulfillment(Φ, ρ, χ, n)</td>
</tr>
<tr>
<td>Recall {Φ, ρ, and χ values}</td>
</tr>
<tr>
<td>Compute Φ = ( \frac{\sum_{j=1}^{m} \mu_j \rho_j \chi_j}{\sum_{j=1}^{m} \varepsilon_j \rho_j \chi_j} ) where; ( \mu_j = \sum_{i=1}^{n} \phi_i \rho_i \chi_i )  ( \varepsilon_j = n(\phi_{\text{max}} \rho_{\text{max}} \chi_{\text{max}}) ); ( n = \text{number of subjects} ).</td>
</tr>
<tr>
<td>Plot/Tabulate ( \mu_j ) vs. ( R_j ); ( { j = 1, 2, \ldots, m } ) used in AP</td>
</tr>
<tr>
<td>End of Function</td>
</tr>
</tbody>
</table>

This function returns a table or a graph of scores \( \mu_j \) vs. requirements \( R_j \). The later gives pictorial view of the level of fulfillment of the requirements.

Calculation of Acceptability Index, α

The χ, φ, and ρ values are the inputs of the algorithm that determines the acceptability of the abstract prototypes [i.e. computation of \( \alpha = \frac{\mu_r}{\varepsilon_s} \), where \( \mu_r = \sum_{i=1}^{n} \sum_{j=1}^{m} \phi_i \rho_i \chi_i \) and \( \varepsilon_s = mn(\phi_{\text{max}} \rho_{\text{max}} \chi_{\text{max}}) \); \( n = \text{number of subjects}; m = \text{number of review criteria} \)]. The even-
tual result is a suggestion on whether the acceptability index ($\alpha$) is within the tolerable limit. The pseudocodes below illustrate how the function that computes the acceptability of an abstract prototype has been implemented.

<table>
<thead>
<tr>
<th>Calculation of the acceptability index, $\alpha$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong> Determine_Acceptability ($\phi$, $\rho$, $\chi$, $m$, $n$)</td>
</tr>
<tr>
<td><strong>Recall</strong> {$\phi$, $\rho$, and $\chi$ values}</td>
</tr>
<tr>
<td><strong>Set</strong> { $\alpha_{thr}$ }</td>
</tr>
<tr>
<td><strong>Compute</strong> $\alpha = \frac{\mu_r}{\varepsilon}$, where $\mu_r = \sum_{i=1}^{m} \sum_{j=1}^{n} \phi_i \rho_i \chi_i$ and $\varepsilon = mn(\phi_{max} \rho_{max} \chi_{max})$;</td>
</tr>
<tr>
<td>($n$ = number of subjects; $m$ = number of review criteria)</td>
</tr>
<tr>
<td><strong>If</strong> $\alpha \geq \alpha_{thr}$ <strong>Then</strong> <strong>Comments</strong> = &quot;Passed $\alpha$-Test - But nevertheless ensure fulfillment of all requirements to a satisfactory level.&quot;</td>
</tr>
<tr>
<td><strong>Else</strong> <strong>Comments</strong> = &quot;Failed $\alpha$-Test - Try other options or improve this alternative and rerun the $\alpha$-test&quot;</td>
</tr>
<tr>
<td><strong>End If</strong></td>
</tr>
<tr>
<td><strong>End of Function</strong></td>
</tr>
</tbody>
</table>

Computation of Relevance Index, $P$

The pseudocodes below describe the algorithm for the implementation that computes the relevance indices ($P$). The inputs into this function are $\rho$ values assigned by the subjects.

<table>
<thead>
<tr>
<th>Computation of the relevance index, $P$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong> Determine_Relevance ($\rho$, $n$)</td>
</tr>
<tr>
<td><strong>Recall</strong> { $\rho$ values }</td>
</tr>
<tr>
<td><strong>Set</strong> { $P_{thr}$ }</td>
</tr>
<tr>
<td><strong>Compute</strong> $P = \frac{\sum_{i=1}^{n} \rho_i}{nP_{max}}$, (where $n$ = number of subjects)</td>
</tr>
<tr>
<td><strong>If</strong> $P \geq P_{thr}$ <strong>Then</strong> <strong>Comments</strong> = &quot;Passed the $P$-Test - Ensure that the corresponding requirement statement is satisfied to an adequate level.&quot;</td>
</tr>
<tr>
<td><strong>Else</strong> <strong>Comments</strong> = &quot;Failed the $P$-Test - The corresponding requirement statement can only be insisted&quot;</td>
</tr>
<tr>
<td><strong>End If</strong></td>
</tr>
<tr>
<td><strong>End of Function</strong></td>
</tr>
</tbody>
</table>

This function returns a proposition, in textual form, on how to treat the individual review criteria when judging to what extent they should be fulfilled.
Calculation of Confidence Index, X

The pseudocodes below show how the function that computes confidence indices has been implemented. The inputs into this function are \( \chi \) values assigned by the subjects.

**Calculation of the confidence index, X**

Function\_Determine\_Confidence (\( \chi \), \( m \), \( n \))
Recall \{\( \chi \) values\}
Set \{ \( X_{\text{thr}} \) \}
Compute \( X = \frac{\sum_{j=1}^{m} \sum_{i=1}^{n} \chi_{ij}}{mnX_{\text{max}}^{\text{criteria}}} \), (where \( n = \) number of subjects; \( m = \) number of review criteria)
If \( X \geq X_{\text{thr}} \), Then Comments = "Passed X-Test - The subject that participated in the abstract prototyping review were confident/knowledgeable/reliable enough"
Else Comments = "Failed X-Test - Re-run abstract prototyping; the subject that participated in the abstract prototyping exercise were not - confident/knowledgeable/reliable enough"
End If
End of Function

This function returns a proposition in textual form, on whether the subjects involved in the AP review were knowledgeable enough about the problem and the review criteria used; or in other words, whether they were confident about the \( \phi \) and \( \rho \) values they have assigned against the review criteria.

**Algorithm for Ranking Requirements**

The relevance (\( \rho \)) values offers means by which the requirements can be prioritized by the representatives of the DSS stakeholders. As mentioned in Chapter 4, the subjects specify, among other parameters, \( \rho \), and by doing so, they also, by default, help ranking the requirements. The requirements prioritization algorithm below has been developed to support the developers in distinguishing which requirements: (i) must be fulfilled unconditionally; (ii) must be fulfilled as completely as possible; and which (iii) are unimportant. This algorithm returns a list of requirements in a salience order.

**Requirements prioritization function**

Function\_Prioritize\_Requirements (\( \rho \), \( n \))
Repeat
\{Recall \( \rho_i \)\}
Compute \( P = \sum_{i=1}^{n} \rho_i \); (\( n = 1, 2, ... \) = number of subjects)
Sort \( \downarrow P_{\text{max}} \)
Sort \( \downarrow \frac{\rho_{\text{max}}}{\rho_{\text{min}}} \)
\}
Return list of requirements in a salience order
End of Function
Algorithm for Supporting the Investigation of what to Improve

It is not always the case that the best solution proposals fulfill all requirements to a satisfactory level. In practice, often the abstract prototypes need to be improved time and again. The function below computes the parameters needed in the determination of the improvement strategy for deficient abstract prototypes. It essentially ranks the requirements based on the collated $\rho$ and $\Phi$ values. With such a ranking, it can be seen how the inadequately fulfilled requirements are relatively judged to be relevant to the software feature under consideration. Once this is known, strategies for improvement of an abstract prototype can be established. Emphasis should be put on ensuring that the requirements with higher $\rho_j$ values and at the same time scored lower $\Phi_j$ values are satisfied sufficiently. It can thus be said that this algorithm systematically takes into account the DSS stakeholders' opinions in the determination of what should be done to improve the abstract prototypes.

<table>
<thead>
<tr>
<th>Improvement strategy function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Function _Determine_Improvement_Strategy} (\rho, \Phi)$</td>
</tr>
<tr>
<td>{ \text{Recall } \rho_j}</td>
</tr>
</tbody>
</table>
| \begin{align*}
| \text{Find } P_j &= \sum_{i=1}^{n} \rho_j; \quad n = 1, 2, \ldots = \text{number of subjects} \\
| \text{Recall scores, i.e. } \mu_j \text{ values} \\
| \text{Compute } \Phi_j, \text{ i.e. } \Phi_j &= \frac{\mu_j}{\mu_{\text{max}}} \\
| \text{Sort } \Phi_j \downarrow \mu_{\text{max}} \\
| \text{When} \\
| \{ \text{Set threshold value } P_{\text{thr}} \text{ for the relevance index} \\
| \text{Set threshold value } \Phi_{\text{thr}} \text{ for the fulfillment index} \\
| \text{Sort } Req = \{(\rho_j, \Phi_j) \\& (\Phi_j < \Phi_{\text{thr}})\} \\
| \text{Return graph and list of requirements with relevance and fulfillment values} \\
| \text{End of Function} |
|} |

Matching DSS Development Task to Existing Requirements

Sometimes the DSS developers are confronted with projects that are similar to those dealt with previously. In this case, there is a great chance that the requirements can closely be analogous. The algorithm below has been designed for use in the identification of the requirements that closely match the problem in hand. This is accomplished by relating pieces of information about the problem in question to the requirements available in the database.

---

9 Refer to Section 4.1.3 for clarification on how the sufficiency is judged.
Problem - requirements matching algorithm

\textbf{Function-Problem-Requirements-Matching} \( (\Pi, \gamma, x) \)
\textbf{For} Problem \( \Pi \)
\{Specify a set of goals/needs \( \Gamma \), where \( \Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_{\nu-1}, \gamma_{\nu}\} \)
\textbf{Search} the database \& match \{\gamma\} to requirements
\textbf{Req} = \{r_1, r_2, \ldots, r_{\nu-1}, r_{\nu}\}
\textbf{Return} list of matching requirements \textbf{Req} \}
\textbf{End of Function}

This algorithm has been implemented in SQL as shown below. It returns requirements model(s) previously used in a similar DSS development projects. In this way, the process of acquisition of requirements for a DSS can be reduced to fine-tuning the already available requirements.

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
\textbf{SELECT} Requirement model, \\
\textbf{FROM} \hspace{1cm} [Analysis Results Database] \\
\textbf{WHERE} \hspace{1cm} (Conditions expressed in Boolean Formulation i.e. AND; OR; NOT); \\
\hline
\end{tabular}
\end{table}

The main advantage is that the time required in gathering requirements for AP can be reduced significantly.

\textbf{5.4.5 Warehouse}

'Warehouse' is a general term used in this dissertation to refer to a repository of data or knowledge used in AP, commonly known as a database or a knowledgebase. Software tools that support data or knowledge input and manipulations are integral parts of the warehouse.

\textbf{Database}

Requirements, representations of the abstract prototypes, and analysis results typically accumulate as the design process of a DSS progresses; and they can be stored for use in the development interval or in future. The information about these entities can be represented formally in text, graphical, audio, video or executable form and stored [Figures 5.8 and A.5 in Appendix A]. The database is designed such that it accommodates several information items related to the entities, and it also embodies local tools for data manipulation. These include software tools to support data input, editing, browsing, and retrieval within the primary database environment. The database is also designed such that several users can share it, and it can be accessed from other modules of the AP software as well. This has been achieved by putting the entire database in a shared folder. This enables users to share the data and use the same forms, reports, queries, macros, and modules. In addition, a provision has been made such that data can be accessed from remote locations, via the Internet as elaborated in Section 5.4.6. This involved creation of Internet Database Connector (IDC) files that allow the user to type queries in an HTML form in order to query a secure Microsoft Access database. The
The main steps to use IDC files to query a database include: (i) using the ODBC\textsuperscript{10} administrator to create a system 'data source name' (DSN) that points to the workgroup information file (System.mdw) used with the secured database; (ii) creation of an HTML form that requests information (the HTML form passes the values to parameters in the IDC file); and (iii) modifying the IDC file to use the input parameters to authenticate user access to the database.

The ODBC data source has been used to connect to data external to Access that does not have built-in drivers. It provides a universal database connectivity application programming interface (API) that enables applications to access data in a wide range of proprietary databases. In the ODBC architecture, an application (in this case Microsoft Access) connects to the 'ODBC driver manager', which in turn uses a specific ODBC driver (for example, Microsoft SQL ODBC driver) to connect to a data source - in this case, an SQL Server database [see e.g. Kruglinski et al., 1998]. Visual C++ tools and Microsoft Foundation Classes (MFC) for ODBC provided means for implementation of separate GUIs for the database engine.

---

\textbf{Linking to the database}

\begin{verbatim}
Function Call Database()
Declarations
    Call the database file
Set the basic application visible
End of Function
\end{verbatim}

\textsuperscript{10} The Microsoft Open Database Connectivity (ODBC) is a standard protocol for accessing information in SQL database servers. In this case, ODBC drivers enable Microsoft Access to connect to SQL database servers and access the data in the SQL databases.
The database provides a background support when the AP software is used, and most functions either receive their inputs from it, or their outputs are destined to it. It avails the necessary data to the developers or the subjects, and facilitates reuse of requirements, abstract prototypes, and information about the problems and their solutions. The function above links the database to the AP software.

Knowledgebase

An advisory utility has been included as part of the AP software to provide general expertise required in testing DSS. Figure 5.9 shows part of the algorithm for selection of testing or review method. There are many review and testing methods in software engineering that can be used in various circumstances and conditions. This algorithm provides a semi-automatic means for selection of an appropriate method. The algorithm for selection of testing or review method has been implemented using CLIPS\textsuperscript{11} shell. Knowledge is represented in CLIPS (see codes below) as a rule, which is a collection of the conditions and the actions to be taken if conditions are met. The inference engine matches the rules to the current state of the system and applies the actions.

**Syntax for representation of rules**

(defrule propagate-goal 
  (goal is ?goal) ;
  (rule (if ?variable $?)
    (then ?goal ? ?value))
=>
  (assert (goal is ?variable)))

**Syntax for representation of facts**

(rule (if type is non-classical and 
  method is no)
  (then category is CI))
  (question method is "Is the purpose to test to requirements?"
  (answer is "I think the best method is " type.method))

The advisory utility has been implemented such that the user responds by simply typing in the "Yes" or "No" answers to questions posed by the software [Figure A.12]. The advisory unit is connected to the AP software via the function below, which calls the form that contains the CLIPS's executable file and the instructions on how to operate it.

**Linking the advisory unit to the AP software**

Function CLIPS_ExpertSystem()

Load Form containing CLIPS file

End of Function

\textsuperscript{11} CLIPS stands for C Language Integrated Production System. It is a rule-based language developed by NASA. Visit http://www.jsc.nasa.gov/~clips or refer to CLIPS Reference Manuals (1997) for more information about CLIPS.
5.4.6 Auxiliary Utilities

The other software utilities that have been implemented and included as integral parts of the AP software include web-based database information manipulation utilities, links

---

12 It must be noted that selection of the evaluation methods was not one of the main focuses of this research. However, the importance of having software support in the selection of the evaluation method was noted, and this algorithm demonstrated the selection concept.
to various web search engines, and a local hypertext information search utility. The AP software is also equipped with the conventional system management utilities, for example, for file management and for providing on-line help.

**Web-Based Database Manipulation Utilities**

The web-based database manipulation utilities provide software tools that allow the DSS developers to work in a distributed environment, and enable subjects in remote locations to access the necessary information via the web. This is achieved as follows. A central database, accessible by all users is provided. The utilities for manipulating this database via the web have been incorporated in the AP software. The function below shows how the web-based database manipulation tools have been implemented.

<table>
<thead>
<tr>
<th>Structure of the web-based database query's source code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;HTML&gt;</code></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>`&lt;FORM METHOD=&quot;GET&quot; ACTION=&quot;Generate-�性_1.IDC&quot;&gt;</td>
</tr>
</tbody>
</table>
| [Specify the functionality:] `<INPUT TYPE="Text" NAME="[Specify the functionality:]">`
| [Specify abstraction level:] `<INPUT TYPE="Text" NAME="[Specify abstraction level:]">`
| `<INPUT TYPE="Submit" VALUE="Run Query">`              |
|                                                       |
| `</HTML>`                                               |

The user can simply type in the requested input information and then click on to run the query. Web-based functions implemented in this way enable the users to execute from remote locations all queries mentioned in Section 5.4.1.

**Using Existing Search Engines**

A tool, which can be used in seeking information when designing DSS, is provided within the AP software. This has been achieved simply by availing URLs known to provide links to the relevant information. The pseudocodes below show how this has been realized.

<table>
<thead>
<tr>
<th>Function linking a web form with URLs to the AP software</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function Web_Search_Engines()</strong></td>
</tr>
<tr>
<td>Get Form size</td>
</tr>
<tr>
<td>Choose/Type in URL address</td>
</tr>
<tr>
<td>Launch Web Browser</td>
</tr>
<tr>
<td>If busy Then</td>
</tr>
<tr>
<td>notify</td>
</tr>
<tr>
<td>Else</td>
</tr>
<tr>
<td>Indicate that it is &quot;Working...&quot;</td>
</tr>
<tr>
<td>End If</td>
</tr>
<tr>
<td>End of Function</td>
</tr>
</tbody>
</table>
Hypertext Information Search Tool

The AP software is equipped with a hypertext utility, which has been implemented by using a special markup language in Rich Text Format (RTF) [refer to Reselman et al., (1999); Craig and Webb, (1998)]. The footnote tags are used to set up the hypertext jumps structure of a file. In the implementation of the hypertext tool, the RTF document was structured such that it contained: (i) topics (written and tagged in the text position of the RTF document and bounded with a break), (ii) jump text, (iii) jump location identification, and (iv) footnotes [which contains jump locations, topic indices and topic subjects]. The hypertext file has been integrated to the AP software via the function below to makes it accessible via the common user interface.

```
Function_Call_Hypertext_Utility ()
  Load Form containing hypertext file
End of Function
```

On-Line Mentor

The AP software comprises an on-line mentor, which in essence provides the user with two utility functions, namely (i) for availing the basic AP concepts to the users, and (ii) for guiding the users in the selection of review subjects and methods of representation of abstract prototypes. These utilities have been implemented by using a special markup language in Rich Text Format (RTF) as explained previously [refer to Craig and Webb, (1998); Reselman et al., (1999)]\(^\text{13}\), and are basically executable files.

5.4.7 User Interfaces

Figure 5.10 shows a schematic representation of the interfaces implemented to provide interaction between the user and the AP software. In general terms, the control over processing is exercised by the user, who issues instructions to the system from the menus. Menus have been preferred for the reason that they have the advantage of being quite natural and of giving the user strong control over the course of decision-making. However, they also have the disadvantage of being slow and rambling, and of being confusing when used in large and complex hierarchical software products.

The AP software required more than one window. Therefore, its user interface has been implemented as a singular multiple-document interface (MDI)\(^\text{14}\), which is the container for all forms mentioned previously. In other words, the MDI provides a parent window from which various functions of the AP software can be accessed and run. The application of the MDI means that the user of the AP software can open many windows and

\(^{13}\) Also visit http://www.mcp.com/info for more information.

\(^{14}\) Refer, e.g. to http://www.microsoft.com/. MDI applications have become very popular way to give uniform appearance to various applications, and many developers find the MDI configuration an easy alternative, and more in-line with the rest of the widely used applications, which generally takes the look and the feel of Web browsers.
applications simultaneously. This brings the users he needed flexibility and convenience in accomplishing the AP activities. For example, during the analysis and interpretation of the results, the developers can simultaneously have, in separate windows on a computer screen, the representations of the abstract prototypes, requirements, analysis results, and all other entities needed when making decisions.

![Diagram](image)

**Figure 5.10: Interfaces of the AP software**

5.5 Conclusion

A pilot implementation of the AP software has been developed based on the concepts presented in Chapters 3 and 4. It is basically an information intensive system, and it facilitates storage and retrieval of: (i) requirements; (ii) information about the abstract prototypes in textual, video, audio, or executable form; and (iii) information about the reviews, namely, the requirements models and the analysis results, in tabular or graphical forms. Although this pilot system as it stands at the moment is not a fully implemented and robust package of software tools needed in AP, it can nevertheless be installed and used in limited real-world situations. The limitations of the AP software are mostly due to the facts that it has been implemented based on the existing applications and it uses an external database management system. One of the cons of using external functions is that in some cases the output(s) of a function may not be transferable to other functions (e.g. because of differences in the data structures). In the situations
where an output of a function cannot be used as input in another function, manual entries are necessary. In general, summoning functions from external applications to perform activities or using an external database makes the AP process to become sluggish, and sometimes operations do not proceed smoothly, at a natural pace. A non-pilot implementation of the AP software should be a self-sustained implementation in that it must be equipped with its own functions and in-house database (rather than an external database management system as it is in this pilot implementation). As for the knowledgebase, it definitely needs further enhancements. The current knowledgebase covers very limited knowledge space. Additional knowledge should be acquired and incorporated into the AP software. It is important to emphasize that the goal of the AP software is not to achieve full automation of the AP process, but rather to provide software tools to assist the developers in AP. The role of humans in AP, especially in decision-making is always vital.
Abstract Prototyping Methodology and Software in Use

This chapter presents the results of the preliminary case studies conducted to explore how AP can be applied in the development of DSS, and how beneficial it can be. First, the method used in the studies is presented, and the case DSS projects involved are briefly described. Then, industrial and research oriented case studies, which demonstrate how to conduct AP reviews by using the AP software are presented and discussed. Afterward, comparative industrial case studies that quantify the validity of the AP concept are presented. Next, the AP scenarios encountered during the case studies are described. Finally, experiences from the case studies are summarized and views of the representatives of the users on the usability of the AP concept are presented. Descriptions of some selected case studies recounting how various AP scenarios can be handled are in Appendix E.

6.1 Method Used in the Studies

Abstract prototyping is a quasi-automated quality assurance strategy developed for use in various DSS development projects. Both the AP methodology and the AP software are generic. To have a general outlook on the strength and usability of the AP concept, two application case studies conducted in a business oriented DSS project as well as in a research oriented DSS project are presented. The specific goals of these studies were threefold, namely to: (i) investigate whether the TMAP paradigm [see Section 4.1.4] is adhered to in the DSS development projects, and if so, how the developers implement it; (ii) study how the AP methodology and software tools can be put in use; and (iii) quantitatively prove that the AP methodology indeed provides the advantages over the conventional procedures.

In using the AP software, first its warehouse must be updated and scaled to the DSS project in question. This means the requirements for the targeted DSS project must first be formulated, sorted according to the abstraction levels, and added into the requirements database¹. One of the key tasks in the case studies had thus been to update the AP software's requirements database by bringing in the requirements for the DSS

¹ The requirements database typically contains generic requirements and/or specifications [see Section 3.6] considered to be applicable to typical DSS projects or software development projects in general. These may be reused in any DSS project.
projects in question. Many requirements acquired in advance [using the scheme presented in Section 4.3] had been sorted, and stored in the requirements database. Numerous abstract prototypes have also been progressively developed, and their representations and the associated information items stored in the AP software's database.

The method used in the case studies can be summarized as follows.

- The developers, that is, either the members of the development team or the other experts who had been entrusted to do parts of the DSS development tasks, conducted the AP reviews at various abstraction levels. The experiments were set out and run by the developers themselves (of course, based on formally defined guidelines).

- Provisional requirements were derived from the database and adapted to the DSS project in question. In some cases, the representatives of the DSS stakeholders were also involved in this adaptation process. The developers created and authored the representations of the abstract prototypes.

- Statistical analyses were carried out to quantitatively attest the validity of the AP methodology as a quality assurance procedure.

- Through a questionnaire survey, by interviewing the developers and the subjects, as well as thorough observations, conclusions on the acceptability of the AP methodology as a quality and trustworthiness assurance strategy for the DSS were drawn.

6.2 Using the Abstract Prototyping Software

As explained in Chapter 5, the role of the AP software at the initial stages of the AP process is mainly to support preparation activities. This includes: (i) supporting acquisition of requirements and authoring the representations of the abstract prototypes; (ii) serving as an on-line mentor by providing expert guidance, for instance, during selection of forms of representation of the abstract prototypes, and in the selection of subjects; and (iii) availing the review information and specific execution guidelines. At the final stages of the AP process, the AP software supports processing of field data. The AP software also allows for requirements, information about the abstract prototypes, and previous analysis results to be stored for use during the design interval or in future projects. In addition, software facilitates the AP process by providing specific templates. A gallery of selected snapshots of GUIs of various utilities of the prototype AP software is included in Appendix A to give an insight on how the AP software can be applied in AP.

To review an abstract prototype of a DSS at a given abstraction level, using the AP software [Figure 6.1], the developers may follow a scenario below, but not necessarily in the order in which the activities are listed [see also Figure 6.2 and Section E.1 in Appendix E].

2 Further descriptions of the activities and steps to follow when implementing AP are available in Appendix E.
- Use the data input utilities to input data in the database [see Figure A.2, A.8, A.19 and A.21 in Appendix A].

- Knowing the 'functionality' and the 'abstraction level', use the MDRS utility in the REQUIREMENTS REPRESENTATION AND PROCESSING module to find the appropriate requirements from the database. Figures A.6 and A.16 in Appendix A show typical requirement processing sessions, by using the MDRS utility.

- Adapt the list of requirements (i.e. omit, e.g., the irrelevant and conflicting requirements, and add the missing requirements) and save the resulting requirements model.

- Transform the requirements into review criteria, i.e. rephrase the requirements into a language that can be understood by the subjects to be involved in the review.

- Use the software tools in the KNOWLEDGE REPRESENTATION AND PROCESSING module to: (i) author from scratch, or search for and edit the relevant representation of the abstract prototype [Figures A.4 and A.3 in Appendix A]; and (ii) save the eventual representation.

- Use utilities in the OPINIONS GATHERING module to construct, format and send the 'Information Gathering Matrix' [Figure A.17 in Appendix A] and the 'representation(s) of the
abstract prototype(s)' to the subjects. In turn, the subjects can submit the completed forms via the web [Figure A.15 in Appendix A].

- Use software tools in the INFORMATION ANALYSIS module to: (i) create the analysis matrix [Figure D.3 in Appendix D]; (ii) assign identifiers to the requirements; (iii) input responses from the subjects, (iv) compute P, X, α and φ indices; (v) generate σ - and φ - diagrams [Figure A.7 in Appendix A]; (vi) rank requirements [Figure A.10 in Appendix A]; and (vii) investigate and decide on the improvement strategy [Figure A.11 in Appendix A].

- Use the cross-referencing utility in the INFORMATION ANALYSIS module to investigate the implications of choosing alternative solutions [see Figure A.18 in Appendix A]. For instance, if a certain theory has been chosen, find other related theories as well as methods, algorithms, and pilot prototypes that must be used alongside the chosen theory. This helps foreseeing beforehand how the chosen abstract prototypes relate to each other.

- The AUXILIARY UTILITIES module houses software tools that support finding requirements as well as possible alternative solutions in the (i) in-house knowledgebase [Figure A.12 and A.15 in Appendix A], (ii) database or knowledgebase of front-end processors, or (iii) in the web [Figure A.16 in Appendix A].

It is important to mention that some preparations must be done beforehand. Such preparations include setting up the rating scales and selection of the subjects. The end product of an AP review at a given abstraction level is a dossier that comprises: (a) the levels of fulfillment of requirements, and (b) a list of alternative solution proposals arranged in the pecking order. The σ - and φ - diagrams provide pictorial overviews of the consistency of the subjects and the degrees of fulfillment of the requirements respectively, while α indices give clues on the levels of acceptability of the abstract prototypes. The qualifying alternatives are then analyzed, modified, and visualized repeatedly until it is assured that all known requirements have been satisfied to a satisfactory level. The decision on whether to keep on improving or to move on to the next abstraction level is taken collectively by the developers, based on their expert judgments.

6.3 Application Case Studies

This section concisely presents the application case studies. It describes the DSS projects involved and the paths of the TMAP paradigm [refer to Section 4.1.4] adopted in various DSS projects, and outlines the advantages gained and lessons learned by exercising AP.

---

3 It must be noted that the information-gathering matrix contains the requirements that must be fulfilled as completely as possible, those whose fulfillment can only be insisted, as well as the requirements that must be fulfilled unconditionally. The later requirements may, however, not be considered during computation of α- and σ- indices.
6.3.1 Industrial Case Study

An industrial case study was conducted at Company Y\(^4\) to validate and quantify the benefits of the AP concept. Activities in this Company are twofold, namely (a) development of software tools to support engineering design and analysis of ships or ship components, and (b) conducting consultancy in ship design. The practice adopted in Company Y is that novel software tools are developed and used first within the firm for some times, and improved, before being marketed and put in service outside the Company. Several software tools have been developed, including those supporting:

- integrated ship design, which includes modules for analysis of ship stability, strength and propulsion characteristics;
- hull form modeling; and
- ship loading and damage evaluation.

Both the designers of the DSS and the programmers in Company Y are experienced ship designers\(^5\). A visit was first made to the Company to introduce the AP concept and to provide guidelines on how to apply it. Literature on this concept was also provided.

Description of the Case Study Project

The case study project was on the development of a module for reengineering of the shape of ship hull. The activities in the process of reengineering of the shape of ship hull include: (i) photogrammetric reconstruction of the geometry of a set of points, (ii) manual construction of topology by indicating connections between points, and (iii) generation of wireframe and B-rep solid models based on the geometric and topological data. The inputs into the prospective module are photographs of specially marked spots on the ship hull, which are in turn processed and used in the regeneration of real world coordinates. The coordinates are subsequently exported to the hull-form modeling module, where computer models of sufficient accuracy and completeness can be created and visualized. The tasks involved in this software development project included: requirements acquisition; design and implementation of a method for estimation of the initial camera locations and orientations; investigation of practical, clear and accurate ways of marking spots on the surface of the ship hull; extension of the manipulation functions in the current user interface; and testing the accuracy of the eventual implementation on the real vessels or sections.

Application of AP

An exclusive interview was carried out after introducing the AP concept to the developers. It revealed that, in most cases, the TMAPE paradigm is in fact intuitively adhered to in Company Y, but the starting point depends on what is initially known, as far as the solution(s) to the problem are concerned. Most frequently, theories or methods

\(^4\) The name of the company is intentionally not revealed.

\(^5\) In fact this is one of the strategies encouraged in software engineering to assure quality and trustworthiness of software [refer, for instance to Kreyman et al., (2002)].
(concealed in the algorithms) are the first solutions that come into the picture during DSS design. Ship design is a very mature discipline, and there are many theories and methods related to various aspects of ship design, often expressed descriptively [see e.g. Koelman, 1999]. Based on these, dedicated algorithms can be created and eventually codes written. Typically this is the TAPE path. Reuse of the existing algorithms (that is, APE path) is also common in this Company. Often algorithms used elsewhere are available, and can be adapted and used in building new ship design support software tools. In this particular case study project, the TMAP path was followed as can be seen in Appendix E [Section E.3.2]6.

To assure quality of the shape reengineering software, requirements were gathered and used as a basis for assessing the adequacy of the underlying theories, methods, algorithms, and as a basis for verification and validation of the eventual pilot implementation. The requirements acquisition procedure [see Sections 3.6.2 and 4.2.1] was followed, and the MDRS software [see Section 5.4.1] was used to support the requirements gathering process. Without following the AP procedure, it would have not been possible to assess the fulfillment of requirements at the theories and methods abstraction levels [see Section E.2.2 in Appendix E]. The AP procedure was also particularly helpful as it enabled the developers to acquire requirements comprehensively in a short time [see the comparative studies in Section 6.3.3].

Concluding Remarks

It can be said that the TMAP strategy is 'unknowingly' a de facto design model in this company. Following the TMAP strategy and working in the framework of AP contributed to quality improvements in that it enabled the requirements to be gathered comprehensively and tested systematically at various abstraction levels. It thus provided a systematic framework for recursive identification and containment of defects as close to their sources as possible. This strategy helped to ensure detection and elimination of defects originating at theories, methods, algorithms, and pilot prototypes abstraction levels (i.e. before passing the DSS design process to the subsequent implementation phase), and it can also be said that it helped to reduce testing costs7.

6.3.2 Research Oriented Case Studies

Case studies were conducted using research Project Z to demonstrate how the AP concept can be put in use and what advantages it brings. This Project is concerned with the development of computer-based theories and methods for shape conceptualization. The anticipated DSS, called Computer Aided Conceptual Design (CACD) software,  

6 It should be noted that by the time of publication of this thesis, the development work was still going on. The underlying theories, methods and algorithms were already developed, and the implementation of a pilot prototype had just begun. It is important to mention that in working in the framework of the AP concept, the developers had the freedom of choosing the appropriate paths (within the TMAP paradigm) to follow.

7 It is advocated that the cost of avoiding an error may be around 1% of the cost of error removal after implementation [refer to, e.g. Jones, (1990)]. In this case, AP reduced the number of errors introduced into the DSS early on, and promoted termination of testing, which according to the literature can easily consume around 40% of the resources [refer to, e.g. Deutsch, (1982)].
supports various conceptual design activities [further details are in the Appendix E]. For each envisaged function, the desirable quality characteristics were specified and the requirement formulated in advance before the development was passed to the design phase. These requirements were grouped according to the abstraction levels at which their fulfillment could be tested at the earliest possible moment. The sorted requirements were linguistically transformed into a language that can be understood by the targeted subjects, and in due course served as the review criteria. The paths pursued [see Section 4.1.4], and the main findings from three application case studies, namely development of a speech recognizer, vague modeler, and 3D points manipulation interface are presented and discussed below. The developers of these components of the CACD software had the duty of marshaling the reviews. They worked within the TMAPE strategy, but they were free to follow any path they deemed to be appropriate.

**Vague Discrete Interval Modeling Software**

Software for supporting artifact designers in vague discrete interval modeling (VDIM) of shapes of products was created. The goal was to come up with an experimental, but 'functioning' modeler, which could be used in exploring whether or not the VDIM theory [Horváth et al., 2000] can be computerized and used in practice. The VDIM software was intended to provide flexible and natural modeling capability, and handle fuzziness, inaccuracies and inconsistencies of input information. Capturing the designer's ideas, as expressed by using natural input means such as gestures and speech was the primary goal. The specific objectives included development of the VDIM theory, describing geometric manipulators, creating the geometric modeling engine (i.e. implementation of a geometric modeling software and a structural manipulation software) and ensuring compatibility with traditional CAD systems. The TMAPE path [Section 4.1.4] was followed. Based on the identified foundational theories such as Boolean and fuzzy set theories; the anticipated designers' actions (i.e. vague modeling methods) were formulated, algorithms developed, and eventually codes written. The Open Inventor software was used as a platform for implementation of the visualization functionality. Some of the existing implementations (that is, parts of the existing codes) in Open Inventor seemed to partly match the problem. In addition, some standard or library algorithms and codes, for instance, for matrix transformations (which are readily available in Open Inventor) were adopted and used. Open Inventor alone, in its natural form [Wernecke, 1994], was not good enough for some sophisticated functions; therefore codes from the Computational Geometric Algorithms Library (CGAL) were used as well. The implementation was tested and the developers felt that it was well suited for the targeted task, and offered faster and efficient vague modeling functionality. It is important, however, to mention that this was just a pilot prototype, for research rather than for commercial purposes. Nevertheless, the developers believed that if fully developed by following the AP methodology, the resulting VDIM software would be acceptable and useful.

In order to assure quality of the VDIM software, the vague modeling theory, the algorithms developed from scratch, and the pilot implementation of the software were reviewed by following the AP procedure. The requirements acquisition method and the
MDRS software were used in gathering requirements, which were in turn processed and used as review criteria [see Section E.2.2.1 in Appendix E]. Reviews were carried out; the levels of fulfillment of the requirements determined, and weak spots in the abstract prototypes uncovered [see Figures E.3, E.4, E.7 and E.8]. Without following the AP procedure, it would have not been possible to measure the levels of fulfillment of requirements at theories, methods, algorithms and pilot prototypes abstraction levels; most probably fulfillment of only one quality characteristic, namely 'functionality', would have been emphasized. This is because the developers revealed that initially the intention was to develop this software with a focus exclusively on this quality characteristic. Thus, there was an imminent possibility of having well functioning software, which nevertheless does not embrace other desirable quality characteristics. The AP procedure was helpful in this respect, as it enabled the developer to acquire requirements that comprehensively described the desirable quality characteristics at theories, methods, algorithms and pilot prototypes abstraction levels. Furthermore, without using the AP software, it could have cost the developer much longer time to acquire requirements.

**Speech Recognition Utility**

The speech recognition utility [Kuczogi et al., 2001] was developed to provide one of the input streams of the CACD system. In the development process, the PAMT - TMAPE path [see Section 4.1.4] was followed. There are several speech recognition systems around that could be applied. The task was thus to adapt a selected (existing) speech recognizer to the envisaged CACD system. The specific goal was to integrate the existing speech recognition software to the CACD system, to define the speech domain, and to make the speech domain extendable. Knowing the goals and the needs, as well as the existing solutions (that is, the most appropriate speech recognition software), a backward path (that is, PAMT) was followed as an attempt to discover the theory behind the speech recognizer. This was needed because the speech recognition theory (concealed in the codes) was not well documented and therefore had to be 'reinvented' and adapted to the current problem, and eventually a dedicated speech recognition software had to be implemented.

The pilot implementation was reviewed in the framework of the AP concept. The requirements acquisition methodology and the MDRS software were used in the collection of requirements, which were then processed and used as review criteria [Section E.3.2.1 in Appendix E]. Data was collected from the subjects and the levels of fulfillment of the review criteria determined [Figure E.8]. The developer confessed that he was initially striving to create an 'effective' speech recognizer. And without following the AP procedure, it was very likely that only one quality characteristic, namely 'effectiveness', would have been taken into consideration seriously. Furthermore, it would have cost the developer much longer time to gather requirements.

---

8 A speech recognizer in this context is software that converts information communicated verbally into text.
The implementation was for research rather than for commercial purposes, but the developer anticipated that if fully developed by taking into the consideration quality requirements brought in by following the AP methodology, a robust verbal control implementation for the CACD system could be realized.

**3D Points Manipulation User Interface**

A graphical user interface was developed to facilitate interaction with points or objects in 3D space. The goal was to have a direct 3D interface to connect the MicroScribe\(^9\) to the Open Inventor rather than a traditional keyboard interface. The implementation was created directly based on theories. That is to say, the TE path [refer to Section 4.1.4] was followed. Having precisely known the needs, and based on the solution ideas, codes were written straight away. There was, however, a need for some sort of algorithmic reasoning. Readily available codes in the Open Inventor were used, as well as the MicroScribe library codes. This implies that the methods concealed in the codes (i.e. the procedures previously translated into codes) were as well adopted and used.

An AP review was conducted after the implementation of the full prototype. The requirements acquisition methodology and the MDRS software were used in collecting requirements, which were in turn used as review criteria [refer to Section E.3.2.1 in Appendix E]. It became evident that it would have been impossible to gather the obtained requirements without using the MDRS software and following the AP procedure. The implementation was already installed and being used when the review was conducted. However, the AP review revealed that the implementation was still lacking some of the desirable quality characteristics [see Figure E.10 in Appendix E].

**Concluding Remarks**

The three studies above were carried out as follow-ups to the industrial case study presented in Section 6.3.1. The general conclusion is that the results match those of the industrial case study. As was for the industrial case study, it has been shown once again that AP helps quality improvement in that it enables requirements to be gathered comprehensively. Furthermore, it also contributes to quality improvement in that it provides a systematic framework for recursive identification and containment of defects as close to their sources as possible. This ensures that defects do not escape the phase in which they are introduced. In this case, defects originating at theories, methods, algorithms or pilot prototypes abstraction levels were detected and eliminated before passing the DSS design process to the subsequent levels. It is universally understood in software engineering that such containments typically lead to high quality software [refer to, e.g., Hevner, 1996]. In addition, it has been shown how the requirements can indirectly be used as metrics for measuring the acceptability of the abstract prototypes of DSS. It has been also been demonstrated that through AP, weak spots in the abstract prototypes can be identified. This provides a way for knowing precisely what lacks, or

---

\(^9\) MicroScribe is a 5 degrees-of-freedom mechanical pointer device, designed to digitize 3D surfaces; it can however be applied as a 6 DoF pointer for graphical interaction as well. (More details are available at www.microscribe.com).
in other words, this alerts the developers what to remedy in order to avert flaws in the final DSS product.

6.3.3 Comparative Studies: Existing Procedures vs. Abstract Prototyping

Comparative studies were carried out in the industrial environment to investigate the influence of the application of the AP methodology on quality improvement and time to market. The DSS project involved was on the development of the shape reengineering software described in Section 6.2.1. This project was large and complex, and its execution was planned to extend over a period of over one year. The studies focused exclusively on the investigation of the influence of AP on quality improvement and on the development time at the requirements specification phase. Due to time limitations in this research and immense capacity requirements, it was not possible to conduct studies on the entire development process.

6.3.3.1 Quality Improvement

This section presents a proof of one of the primary hypotheses of this thesis, namely, the application of AP can remedy the problem of lack of adequate requirements in DSS development projects and improve quality of DSS. It is known that quality in software development means satisfaction of requirements or implied needs [see e.g. Jorgensen (1999); Xenos and Christodoulakis (1997)]. In other words, requirements are the software quality agents, and if the requirements for the software are not defined comprehensively, it is difficult to achieve the desired level of quality. It follows that a DSS, like any other software, would be of high quality only if its requirements are comprehensively specified. This section analyses how the application of AP assisted the developers to specify requirements comprehensively.

Quantitative Analysis of How AP Contributed to Quality Improvement

Generally, there is no universally accepted method for measuring software quality. In fact, there are several methods that can broadly be grouped as direct or indirect methods. Direct measures such as defects and reliability measures [refer to e.g. Tian et al., 1997] can be used to assess quality of software, while on the other hand, software quality indicators and factors, such as the number of requirements, can be used to indirectly measure or implicate software quality [refer to, e.g. Jorgensen, 1999]. In this work, the contribution of the AP methodology to quality assurance of the DSS is measured indirectly, namely, by comparing statistically the number of requirements gathered by following the AP procedure vs. the number of requirements specified by following the conventional requirements gathering techniques.

The developers first gathered requirements as they normally do, and later on they were asked to follow the AP procedure. They gathered requirements by following the

\[10\] Publications indicate that lack of adequate requirements is currently one of the main reasons for poor quality of software or failures of complex software project [see e.g. Standish Group Report (1994), NIST Report (2002)].
conventional procedures\textsuperscript{11} such as needs analysis and brainstorming, and eventually came up with a total of 25 requirements. Out of these 25 requirements, combinations of 7, 9, 21 and 18 requirements testable at theories, methods, algorithms and pilot prototypes levels respectively were formulated. The AP procedure summarized in Section 6.2 was then followed, and the methodology for preparation of the requirement models [refer to Sections 3.6.2 and 4.2.1] was applied in the acquisition of requirements testable at theories, methods, algorithms, and at the pilot prototypes abstraction levels. The MDRS software [see Section 5.4.1] was used to find the existing requirements (i.e. from the AP software's database) that could be reused in this task. The requirements derived from the database were then sent to the developers, who reviewed and adapted them to this work. Most of the requirements were reused unchanged, and some of them were dropped outright. In the end, the number of requirements grew to 104. Out of these 104 requirements, combinations of 24, 29, 43 and 39 requirements were testable at theories, methods, algorithms, and at the pilot prototypes abstraction levels respectively. These requirements described, in various abstraction levels, how the shape reengineering software would be like. They were then: (i) transformed and used as review criteria [Section E.2.2 in Appendix E], and/or (ii) as a checklist for verifying whether or not the desirable quality characteristics are being taken on board during the design of the shape reengineering software.

Figure 6.3 summarizes the statistics on the requirements. As can be seen, the number of requirements increased significantly as a result of AP. It is universally understood that as more requirements are taken into consideration, there is a better chance of developing high quality software.

![Figure 6.3: Comparison of the numbers of the requirements gathered with and without AP](image)

Figure 6.4 compares the number of the requirements formulated without following the AP procedure with those gathered by following the AP procedure. Only 24% of the total number of verified requirements was formulated by following the conventional requirements acquisition methods. The rest came as a result of the application of the AP methodology. Thus, it can be said that

\begin{center}
\textbf{Figure 6.4: Comparison of requirements acquired with and without following the AP procedure}
\end{center}

\footnotesize{\textsuperscript{11} Refer to Section 3.6.2 for more information about the procedures used in the acquisition of requirements.}
without following the AP procedure, up to 76% of quality requirements could have never been taken into the consideration\textsuperscript{12}. This would have had vast adverse effects on quality of the shape reengineering software.

Figure 6.5 shows the proportions of the requirements formulated without following the AP procedure in the requirement models at the four abstraction levels. It can be seen that they made up only less than 50% of the total numbers of requirements in the requirements models. This highlights the importance of the application of AP and the significance of software support in the acquisition of requirements for the abstract prototypes. Another observation from the histogram is that most of the requirements formulated without following the AP procedure were testable at the algorithms abstraction level and after the implementation of a pilot prototype. This can be attributed to the fact that the developers didn’t think about the software from theoretical and methodological points of views, and it was therefore difficult to come up with sufficient number of requirements testable at these abstraction levels. In that case, there was an imminent danger of developing the shape reengineering software based on flawed or even insufficient theories or methods. Working in the framework of AP reduced this possibility.

Figure 6.6 shows the percentages of the requirements testable at various abstraction levels. It can be seen that only 23.1%, 27.9%, 41.4% and 37.5% of the total number of requirements could be tested at the theories, methods, algorithms, and pilot prototypes abstraction levels respectively. This proves

\textsuperscript{12} However, it is important to mention that this large percentage is partly attributed to the fact that in practice, the developers often tend to take some of the requirements as obvious, and they usually do not formally specify 'obvious' requirements. The problem, however, as pointed out in the literature [refer to, e.g., Moore, 2000] is that there is always the danger of not considering requirements if they have not been specified formally.
one of the fundamental hypotheses of this research work that in order to achieve the acceptable level of quality, reviews and/or tests should be conducted at all four abstraction levels. Not adhering to this can simply be regarded as partial review or testing, since, as it can be seen, quality cannot be assured in one or some selected abstraction levels exclusively.

Figure 6.7 shows the proportions of the requirements that could be tested exclusively at theories, methods, and algorithms abstraction levels, and not at any other level, even after the implementation of full prototype. Of the total number of verified requirements, 10.6%, 9.6%, and 21.2% could only be tested at the theories, methods and algorithms abstraction levels respectively. This means that if reviews or tests were not carried out at the theories abstraction level, 10.6% of quality factors would have never been taken into consideration. Similarly, if reviews or tests were not conducted at the methods and algorithms abstraction levels, 9.6% and 21.2% respectively of quality factors would have never been taken into the consideration.

Apart from applying AP in the development of the shape reengineering software, it has also been applied in four other DSS projects or portions of DSS projects [Section 6.3.2 and Appendix E]. Figure 6.8 shows the statistics on the application of the AP methodology in these projects. It can be seen that most of the requirements in the DSS projects were formulated as a result of following the AP procedure. Based on these statistics, it can be said that gathering requirements by following the AP methodology
(while supported by a searchable database that contains good requirements\textsuperscript{13}) enables the envisaged DSS to be described comprehensively.

**Functional Improvement**

Comprehensive specification of requirements is only one way of showing that AP improves quality and performance of DSS. An unequivocal way would be to compare in practical situations the performances of two actual implementations, namely, an implementation developed by following the AP procedure vs. an implementation developed without following the AP procedure. However, rerunning the development process in these two fronts would have required substantial budget, resolve and sacrifice on the part of the DSS development firm involved in this study. Time limitation was another constraint. Hence, it was impossible to quantify the contribution of AP on the improvement of the performance of the shape reengineering software, by studying the actual performances. However, the improvement of the performance of the eventual DSS can, at least, be predicted by analyzing the influences of the requirements. The requirements that were formulated by following the AP procedure such as: 'The rounding or truncation errors should not accumulate to destroy the answer'; and 'The algorithm should not halt without notification' [refer to Table E.8] should result in improvement of quality of the shape reengineering software. Similarly, consideration of the requirements such as 'Structured output, i.e. no 'point cloud'; 'The number of control points needed should be as small as possible (as small amount of photographs as possible)' or 'It should be suitable for small and large vessels' [refer to Tables E.6 and E.7] early on influenced the selection of theories, methods, and the design of the shape reengineering software; and should also influence its functional characteristics.

**Summary and Concluding Remarks**

The statistics above underscore the significance of the AP methodology. Without following the AP procedure:

- Only 24\% of the requirements would have been tested or taken into the consideration, and up to 76\% of the desirable quality requirements could have never been considered during the development process. This can partly be attributed to the fact that the problem was complex, and with the prevailing requirement acquisition procedures, it was difficult to formulate quality requirements exhaustively. Specification of what is required from theoretical, methodological and algorithm perspectives in a way helped to rectify this. As a result of following the AP methodology, the requirements increased three times. Consideration of the additional requirements vastly boosted the chance of developing quality shape reengineering software.

- Testing could have waited until after having a tangible implementation (i.e. after coding); and this according to the statistics above means that:

\textsuperscript{13} Typical characteristics of good requirements are in Appendix B.
only 37.5% of the requirements could have been considered or tested [Figure 6.6]; and

up to 41.35% of the requirements could have never been considered or tested\textsuperscript{14} [Figure 6.7].

- In the worst case, wrong theories, or flawed methods and algorithms could have been deployed.

In conclusion, it can be said that the AP concept contributed to quality improvement since it helped to expose additional quality requirements that would have otherwise never been taken into the consideration. The statistics above indicate that through AP, many additional quality requirements can be specified, and naturally this increases the chance of developing quality DSS. This is due to the fact that requirements describe what the DSS will be like, and by clearly describing the features and the desirable functional and non-functional characteristics of the DSS, quality can be assured. Furthermore, in the opinions of those who worked in the framework of AP, it enabled them to work systematically at the design phase [Section 6.3.5]. And this in turn helped to assure quality of the shape re-engineering software\textsuperscript{15}.

6.3.3.2 Influence of Abstract Prototyping on the Development Time in the Requirements Specification Phase

The time a DSS takes in any of its development phases contributes to the time to market. In general terms, savings can be made if the times spent to accomplish activities in various phases are shorter. This section presents a quantitative study conducted to compare time spent in acquiring requirements by following the traditional approaches against time spent in gathering requirements in the framework of the AP concept. The method used can be described as follows. The requirements were gathered first by following the traditional procedures, and later on by following the AP procedure. The times taken to gather requirements in each case were recorded. There was no overlap between the two processes. Precaution was taken to ensure that the developers weren’t aware that this study was going on in order to avoid letting them work in an unusual pace. Table 6.1 presents the time spent and the acquisition rates for the two cases.

As it can be seen, the requirements acquisition rate was significantly higher when working according to the AP concept. Normally, the requirements acquisition process is a very involving and time consuming process, and extends to, for instance, holding series of brainstorming and consultative meetings, literature studies, and so forth. Gathering requirements for the shape reengineering software without following the AP procedure was no exception. The conclusion from this analysis is that the application of

\textsuperscript{14} Without AP, reviews or tests could have never ever been conducted formally at the theories and methods abstraction levels, and it is assumed that not all specified ‘algorithmic’ requirements could have been addressed.

\textsuperscript{15} It is well understood that assurance of the process by which software is developed is an essential requirement in quality assurance [refer e.g. to Petschenik, (1985)].
the AP methodology and software reduces the time required in the acquisition of requirements, and therefore shortens the time to market.\textsuperscript{16}

\begin{table}[h]
\centering
\caption{Time spent on the acquisition of requirements}
\begin{tabular}{llll}
\hline
Acquisition method & Number of requirements ($N$) & Time ($t$) taken to acquire the requirements [man-days] & Acquisition rate ($\frac{N}{t}$) [requirements/man-day] \\
\hline
Traditional approaches & 25 & 11 & 2.27 \\
Abstract prototyping & 79 & 4 & 19.75 \\
\hline
\end{tabular}
\end{table}

6.3.4 Abstract Prototyping Scenarios

Three situations prevailed at the onset or at any other stage of the DSS design process. Either multiple, only one, or no solution situation scenarios prevailed. The 'multiple solution' scenarios refer to situations whereby several theories, methods, algorithms or existing codes were available, and the developers had to evaluate each alternative and select one of them. The 'single solution' scenario refers to circumstances in which the developers were obliged to deploy a certain theory, method, algorithm or existing codes (for instance, due to technological constraints). There were also cases in which there were 'no existing solutions' that matched the problems at all, and the developers ought to create solutions from scratch. This section outlines, by referring to various real world representative case studies how these three scenarios were dealt with in AP.

'Multiple Solution' Scenarios

For the problems that more than one solution was known, AP repeatedly involved: (a) finding the highest-ranking alternatives, (b) enhancement (if the alternatives were of low grade), (c) combining (i.e. adding, separating, overlapping) with other approved solutions, and (d) reevaluation. The procedure can be summarized as follows. Requirements were formulated and transformed into review criteria. Views of various stakeholders on whether the scenarios met the review criteria were collected. The acceptability indices ($\alpha$) of the alternatives were subsequently computed. The best alternatives were those with the highest values of $\alpha$. $\phi$ - and/or $\sigma$ - diagrams for the best alternatives were then plotted to determine if there were any weak spots. If $\alpha$ for the best alternative happened to be greater than the threshold value ($\alpha_{thr}$) and there were no significant weak spots\textsuperscript{17}, then the alternative was considered suitable, and work in the subsequent abstraction level could be started. Otherwise the alternative was considered

\textsuperscript{16} It should be noted that it is estimated that 40\% of the software development efforts is put into requirements gathering and analysis [NIST Report, 2002].

* One man-day is equal to an eight-hour working day.

\textsuperscript{17} A 'no significant weak spot situation' prevails if the $\phi$ – or $\sigma$ – diagram shows approximately even levels of fulfillment of requirements.
to be unsuitable, and was improved and reevaluated repeatedly. The second best and other high-ranking alternatives were considered as well. In the situations where few high-ranking alternatives had closer $\alpha$ values, it was helpful to granulate review criteria and/or to consider extra quality aspects, for instance, those related to the subsequent development phases. This helped the developers arrive at a more precise conclusion. The AP reviews presented in Sections E.2.1.2.1 (Case I), E.2.1.2.2, and E.2.1.2.3 give insights on how the 'multiple solution' scenarios have been treated.

'Single Existing Solution' Scenario

For functions that only one solution was known, AP recursively involved: (a) identification of weak spots, (b) improvement of the solution, (c) fitting the solution to ensure that it works properly alongside other related (approved) solutions, and (d) reevaluation. The $\Phi$ indices provided means for determination of the weak spots and for comparing the solution in hand to an ideal solution. An ideal solution implies fulfillment of the requirements to the highest possible extent (that is, $\Phi = 100\%$ for all requirements). However, practically this was never the case, as different subjects had different views, preferences, and mentality in assigning the scores (for instance, some assigned generous values of $\phi$, $\rho$ and $\chi$ when filling the information gathering matrixes while others did otherwise). The application case study presented in Section E.2.1.2.4 illustrates how the 'single solution' scenarios were dealt with.

'No Existing Solution' Scenario

A 'no existing solution' scenario can happen at any abstraction level. For any problem that no solution was known in advance, a novel solution was created. The developers themselves created the solutions in the form of theories, methods, algorithms or pilot prototypes; and the users could also be involved in this. The AP procedure was then applied to determine the adequacy of the created solutions. The $\phi$ - and/or $\sigma$ - diagrams were subsequently plotted, and $\alpha$ index determined. If there were no notable weak spots, then the solution in question was regarded as adequate, and activities in the subsequent level of implementation could be started. Otherwise, the solution was considered to be unsuitable and consequently strengthened and reevaluated. The case study presented in Section E.2.1.2.1 (Case II) demonstrates how the 'no existing solution' scenario situations were administered.

Handling of the Individual Comments and Suggestions

Very often there were individual comments and suggestions given by the subjects during AP. In general terms, there are no objective criteria or universally accepted procedures for handling the individualistic views. Thus, simple strength - weakness analyses were conducted to judge whether or not to take the suggestions on board. Basically this involved stipulation and analysis of the pros and cons of the suggestions. It provided the foundation for distinguishing opinions of outstanding quality from mediocre ones. Expertise, knowledge and experience had been the key assets in dealing with the individual comments and suggestions.
6.3.5 Verdict of the Users of the Abstract Prototyping Concept

A questionnaire survey was carried out to gather opinions of the developers and various subjects involved in all case studies. The aim was to investigate the acceptability of the AP concept. They were asked to indicate how AP helped or supported various aspects. A four-level rating scale of 0 to 3 for not effective, fairly effective, effective, and very effective respectively was adopted. Eighteen of the subjects responded, and Figure 6.9 shows the statistics of the collected views. It can be seen that in the opinions of the developers and the subjects who participated in various case studies, AP emphatically supported: (i) shaping and directing activities at the design phase, (ii) pre-implementation testing, (iii) identification of weak spots, and (iv) multi-stage design, review and measurement of fulfillment of requirements.

![Figure 6.9: Views on the acceptance of the AP concept](image)

The collated data also indicate that most of the canvassed developers and subjects resoundingly felt that the AP procedure is worthwhile in supporting acquisition of requirements for the abstract prototypes, and in the evaluation of alternative implementations and selection of the most appropriate option. Furthermore, both personal testimonials and systematically collected data showed a general consensus that the adopted metrics and measurements provided reasonably objective results. Methods for acquisition of requirements, selection of alternatives, and for measurement of fulfillment of requirements in AP are dependent on techniques adopted from other fields such as requirements engineering and software measurement, most of which are still subjects of intensive research. It is however, important to mention that the statistics in Figure 6.9 are just indicative, and may not accurately reflect the collective opinion of

---

18 This rating scale was tentatively chosen to get 'quick-and-dirty' feedback on the acceptability of the AP concept. In order to obtain more accurate results, a finer scale must be adopted.
the DSS developers’ community. Clearly, they by no means give conclusive indication on the acceptability of the AP concept. Much more studies still need to be conducted in a variety of real world applications to come up with conclusive results.

6.3.6 Summary and Lessons Learnt

The AP methodology has been applied both in a well-defined commercial DSS development project (in which consistency, completeness and business related requirements such as schedules and cost overrun were indispensible) and in a research oriented DSS development project (where novelty and contribution to knowledge were important). These studies gave impression on how it can be used in these two major DSS development environments. The results of the studies suggest that the AP methodology can generally be used in the development of any DSS successfully. However, further studies need to be conducted to validate the AP concept in various practical situations and to explore more about its potentials.

AP served primarily as a model for directing the development and the review activities at the design phase. In this case: (i) creation of theories; (ii) formulation of methods; (iii) design of algorithms; and (iv) implementation of pilot prototypes were regarded as the design activities. Products of each activity (i.e. theory, method, algorithm, or pilot prototype) were reviewed thoroughly (i.e. against an exhaustive list of review criteria derived from requirements) before passing the process to the subsequent stage. Several alternative solution proposals often came into the picture at the higher abstraction levels, namely, at theories and methods levels. These were reviewed, flaws detected, and the best alternative(s) selected. Algorithms and pilot prototypes are also testable intermediate implementations (in the context of the AP concept), and are as well among the commonly recognized intermediate implementations when following traditional software development models such as the waterfall and the program growth model. These intermediate implementations were reviewed as well in the framework of the AP concept.

There were, however, other application avenues and dividends of AP. For instance, as witnessed during the case studies, AP served as a technique for keeping the requirements aboard during design. The MDRS utility [refer to Section 5.4.1] provided powerful means for searching and cross-referencing the requirements to functions and the abstraction levels. In this way, it could effectively be traced, for instance, whether a requirement has or hasn’t been fulfilled, who requires it, and so forth. AP was also deployed as a technique for acquiring and managing information about the abstract prototypes, a methodology for involving various stakeholders in the DSS design process, and as a framework for systematically introducing methods and DSS into the industry. Bringing into the review floors (and indeed into the DSS design process) the representatives of future users to serve as members of the review panels manifests the later application orientations. This, by default, builds up and consolidates relationship between the developers and the eventual users early on. It is also tempting to suggest, based on the results of the application case studies that the AP concept can be used differently or even outside its intended domain (i.e. development of DSS), for example, in the development of other software that model or predict physical phenomena.
The main lessons learned from the application of the AP methodology in its targeted application domain (i.e. in the development of DSS) can be summarized as follows:

- In all application cases, the Theory-Methods- Algorithms-Pilot prototypes paradigm was intuitively followed, but not necessarily in a strict singular order [Table 6.2]. The tracks pursued largely depended upon in what form the solution was initially available.

<table>
<thead>
<tr>
<th>DSS project</th>
<th>Paths followed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speech recognition utility</td>
<td>PAMT/ TMAPE</td>
</tr>
<tr>
<td>VDIM software</td>
<td>TMAPE</td>
</tr>
<tr>
<td>Shape reengineering software</td>
<td>TMAPE</td>
</tr>
<tr>
<td>3D points manipulation UI</td>
<td>TE</td>
</tr>
</tbody>
</table>

- The application cases have attested that the AP methodology can effectively help the DSS developers in the design phase to: (i) work in an orderly and guided manner, (ii) quickly and thoroughly acquire requirements for the entire DSS as well as for the abstract prototypes, and (iii) systematically review the abstract prototypes, by involving various DSS stakeholders as the review subjects. It has been shown that the AP procedure provides a framework for determination of the adequacy of the abstract prototypes and helps in the identification of weak spots.

- Solid understanding of the foundational engineering principles was always necessary when designing DSS. This was always the case, even for problems that had high-fidelity initial solutions. Algorithms or codes for the building blocks of the DSS were adapted and utilized only after thorough analysis, and sometimes this involved recalibration of the underlying theories or methods.

- Involvement of the users in the reviews at the 'methods' abstraction level, especially in highly technical and scientific DSS projects, was seen by some developers as rather unfeasible. This was because introducing highly technical and scientific concepts to clients (who are typically rather novice, as far as the technical details are concerned) required a lot of efforts, and sometimes it was difficult to make them understand such concepts to the level of making meaningful reviews. For such projects, the involvement of the users in AP had to be delayed until after the implementation of the pilot prototypes.

- The requirements for the abstract prototypes were highly transferable from task to task, and from project to project. In most cases, the peer level general-purpose requirements were used unchanged in various AP review exercises. The requirements for the research-oriented projects were formulated first, and the overwhelming numbers of the general-purpose requirements for the abstract prototypes were used unchanged in the industrial case study.

- Some of the developers who worked in the framework of the AP concept could not immediately understand what a theory or a method means, or differentiates theories from methods straight away. Often it was necessary to provide definitions. Perhaps, this is due to the fact that theories and methods are not commonly recognized as tangible intermediate implementations in the existing software development models.
Products of the later DSS design stages (namely, algorithms and pilot prototypes) were, however, easily distinguishable.

- There have been skepticismis on the reliability of AP metrics and measurements. In measuring the acceptability of the abstract prototypes or the extent to which they fulfill requirements, some users questioned the objectivity of the final results, and singled out the composition of the review panels and completeness of review criteria as the main reasons behind their skepticism.\(^{19}\)

- The AP concept appeared to work slightly more convincingly in the industrial DSS project than in the research oriented DSS project. This can be attributed to the nature of the projects. In the industrial DSS project, the requirements were much clearer and testable, while in the research oriented DSS project, the requirements were somewhat vague, and some of them were rather difficult to test. Nevertheless, it can be said that the philosophy of AP was evenly appreciated in the research oriented as well as in the industrial DSS project.

### 6.4 Conclusion

The AP methodology and software tools have been applied in the processes of development of DSS. Most developers and subjects who worked in the framework of the AP concept unanimously deemed it as a logical way of designing and ensuring quality and trustworthiness of DSS. It has been demonstrated through the application case studies that AP: (i) offers a scheme for shaping and directing the DSS design activities, (ii) allows for exhaustive collection of requirements, (iii) provides a systematic method for involving various stakeholders in checking progressively if the requirements are being met satisfactorily, and (iv) equips the developers with a framework for testing and measuring how well the intermediate implementations at the design interval are being developed. The methodology provided a foundation for systematic identification of weak spots in the abstract prototypes, a scheme for selection of the best alternatives through the consensus of the representatives of the stakeholders, and offered means for early recognition of the needs for enhancements. In relation to requirements engineering, AP: (a) facilitated the acquisition of requirements; and (b) provided means for measuring how well the requirements have been fulfilled, and for establishing priorities that need to be given to various requirements. As a stakeholders-centered pre-implementation testing strategy, AP allowed for various abstract prototypes to be reviewed, and for the stakeholders of DSS to be involved in this. It can be said that it promoted consideration of the aims, possibilities, and various quality characteristics in a systematic way early on, and reduced imperfections that often lead to a need for rework when software happen to be poorly engineered. In other words, AP reduces susceptibility of the abstract prototypes to errors, and it averts the possibility of institutionalization of bad decisions, which can otherwise jeopardize the DSS

---

\(^{19}\) In general terms, skepticism on metrics and measurements is common in many software development projects or organizations. Nevertheless, it is understood that without metrics, it is impossible to know how well various products have been implemented [refer to Section 2.5.5].
development project, or adversely affect quality and trustworthiness of the eventual DSS. As a result of the application of this technique, proven and reliable theories, methods and algorithms can be used in the creation of the DSS. Exercising AP can ultimately warrant realization of usable and low risk DSS.

It can be said that the case studies have vindicated the validity of the AP concept, and have not revealed any result that invalidate the research hypothesis stipulated in Section 1.4. In fact, the studies proved that the DSS design process indeed passes through various stages; implicating unique abstraction levels and contexts, and yield various different testable intermediate implementations as speculated. It has also been shown that many additional requirements can be gathered by following the AP procedure. Based on these requirements, it was possible to spot faults in the abstract prototypes early on and in an orderly manner, and various DSS stakeholders have systematically been involved in the defects detection process. The AP methodology is adaptable, and the AP process is phased and recursive so as to allow for containment of defects close to their sources. Those who applied it indicated that it is quite effective and its application contributes in the improvement of quality and trustworthiness of DSS, and warrant their acceptance.

Skeptics might hasten to point out that what have been presented in this chapter are merely very limited and controlled case studies, which involved dismal mix of DSS development projects. While this may be the case, it is also true that the results vindicated the usefulness of the AP concept. And it is tempting to suggest, based on the achieved results that the AP concept can be applied in many other DSS projects and do well. The underlying methods are project independent, while the AP software can be tailored to meet specific demands, by just furnishing its warehouse with appropriate knowledge and requirements, without the need for any major structural modification or extensive coding. However, for the AP methodology to be even more successful, it needs to be integrated with suitable metrics, and applied in the framework of an appropriate software process model. It should also be complemented with traditional software tests.
Chapter 7

Discussion, Conclusions and Future Work

This chapter presents an overview of the research work and discusses the results, gives general conclusions, and presents recommendations for future work. It is structured as follows. First, the main findings of the research work are concisely presented, and the implications of the results are discussed. Afterwards, the contributions of the research and limitations of the achieved results are presented. Finally, some recommendations for possible future research directions are outlined.

7.1 Findings of the Research and Implications

The activities in this research have been threefold, namely: (a) studying the processes of development of DSS and concluding on a concept for pre-implementation testing of DSS; (b) creation of a computer based method for pre-implementation testing of DSS; and (c) application of the pre-implementation testing concept in real world situations to investigate its validity and how it can be put in use. The results of these research activities are revisited and discussed in the following sub sections.

Processes of Development of Design Support Software

The processes of development of DSS have been analyzed. Like for other software tools, they include requirements analysis and specification, design, implementation, testing and operation as the main phases of the development process. Many publications rallied to point out that concepts, models and verification and validation methods used in software development address requirements, designs and codes as the in-process implementations in the development interval. Faults in the codes can be traced back to the requirements or designs. Unlike some software used in other professions, DSS are based on engineering principles and are typically complex. Furthermore, the design phase in the process of development of typical DSS is broad and yields distinct intermediate implementations. It has been established through literature and empirical studies that the main activities in the design phase include: (i) assignment of features to the imagined DSS system, (ii) selection or development of foundational theories for each feature, (iii) formulation of methods, (iv) design of algorithms, and (v) writing codes for pilot prototypes. The initial activities, namely, development of theories and methods are most often delegated to research institutions, for instance, as contract research, but
sometimes the R&D departments within software companies are entrusted to accomplish these activities. The later activities are traditionally done in the mainstream departments of software development companies. The above-mentioned activities yield four in-process implementations that can formally be reviewed or tested, namely, theories, methods, algorithms and pilot prototypes.

Surely, there is a gap between what the existing software development models and testing techniques offer, and what is actually required in the development of DSS. It is indispensable that the intermediate implementations in the design phase should be treated just like other intermediate implementations in the DSS development process, and formally reviewed or tested. The benefits of reviewing or testing the in-process implementations can clearly be seen. Even common sense suggests that recognition of these intermediate implementations as testable, and testing them like other intermediate implementations could be instrumental in assuring quality and trustworthiness of the prospective DSS. Another advantage is that, by having clear goals and deliverables in the design phase, the developers are forced to consider the intermediate requirements. They are also made to think about alternative solutions at various abstraction levels, analyze them, select the most prominent one and improve it repeatedly. As a result, flawless intermediate implementations can be realized. This can avert many problems. For example, the need for rewriting codes if the software is poorly engineered can be avoided. Other advantages include: (a) providing opportunity for review and early enhancement of DSS, (b) paying attention to the DSS project's goals continuously, and (c) supporting synthesis of partial solutions and allowing for various different combinations of proposals to be reviewed. It is also tempting to presume that a continuous review of the implementations in the early phases of the DSS development process, in particular those achieved before coding can have a huge impact on time to market and costs, since the development track is assured early on. However, it should be noted that application of a new additional review method might necessitate additional overheads, increased workload and possibly new resource requirements. It is also universally understood that formal methods limit creativity to a certain extent. On the other hand, time consumption with regard to usage of an additional method is generally uncertain. The application of AP might altogether slow down the DSS development process\textsuperscript{1}. Trade offs are typically necessary, and at the end of the day, the effects of using additional method should outweigh those of not using it.

Abstract Prototyping Concept

The AP concept has been introduced as an attempt to fill up the existing gap and reinforce quality assurance in the early stages of DSS development. The concrete purposes of the AP concept can be summarized as to:

- capture and to record requirements for the intermediate implementations of DSS in the design phase;

\textsuperscript{1} This is also true with regard to the application of other formal methodologies [Refer e.g. to Schluter et al., 1999].
provide measurements and checkpoints for the developers in the design phase;

- enable various stakeholders including outside experts and future users to be involved systematically in the DSS projects early on;

- promote both creativity and systematic thinking during DSS design;

- reduce complexity and emphasize on specific and essential characteristics of the problem at a time;

- systemize the DSS design process such that the milestones are explicit, and the initial inputs from which the developers can start deriving concrete solutions are formally known;

- support the investigation of related knowledge in building DSS and taking into account multitudes of aspects and opinions;

- force planning of the DSS design process, and enable systematic execution of the DSS design activities; and

- provide a framework for concurrent development and testing, deployment, and preparation of long-term support facilities, in particular, when applied in contracted DSS development projects.

It can be said that AP furnishes the DSS developers with a theoretical foundation for quality assurance in various different contexts. It introduces various in-process implementations, called abstract prototypes that are not recognized in the framework of the conventional software development models. These are theories, methods, algorithms, and pilot prototypes. AP can be regarded as an alternative to working within the traditional software development models [Figure 7.1]. However, in this case, the DSS design process has clear milestones rather than just proceeding in an unstructured fashion, without formal in-process milestones. AP is a vicious procedure in that candidate theories, methods, algorithms, or pilot prototypes are reviewed first, and then various combinations are reviewed as well. The understanding in this work, however, is that, ensuring quality of these abstract prototypes cannot necessarily guarantee quality of the eventual DSS. To avoid the remnant oversights, it is necessary to conduct traditional tests such as unit, system, integration and usability tests after implementation of the entire software. The reason for this is that some of the requirements cannot be tested at all within the abstraction levels. Conversely, some of the requirements cannot be tested after coding as well. This is because the DSS typically houses theories, methods and algorithms, and sometimes it is difficult to see if the right theories, appropriate methods, or good algorithms have been deployed by just testing codes only. It is thus equally important to ensure that flawless theories, methods and algorithms are used.
Figure 7.1: Position of AP in the DSS development

Formal Methods

Formal methods for acquisition of requirements, representation and processing of data that describes abstract prototypes, opinions gathering, and for measuring how well various abstract prototypes are being developed have been defined. AP offers the DSS developers community the following:

- A model for directing and shaping activities at the DSS design phase.

- A formal procedure for prototyping and conducting reviews without necessarily using executable codes.

- A framework for representation and processing of requirements. It encompasses formal procedures for acquisition of requirements and sorting them according to the abstraction levels, and mechanisms for: (i) searching and selecting requirements that match a task in hand, and (ii) for crosschecking requirements during the design period.

- A scheme for selecting solutions based on the opinions of the representatives of various stakeholders. However, the accuracy of the results of the selection process largely depends upon several factors such as completeness of the requirements, reliability of metrics and measurements\(^2\), and sincerity of subjects. Right selections in the early phases of the DSS development lets the developers avoid investing efforts,

---

\(^2\) Ensuring the completeness of requirements or validity of the adopted metrics and measurements has not been among the main subjects of this research. The metrics and measurements used in this work and the review methodology in general are experimental, and may need to be recalibrated. On the other hand, as mentioned in Section 4.2.4, users of the AP concept can altogether device their own measurement methods or even create their own evaluation schemes.
time and resources on ideas or designs that would later be rejected for not meeting the acceptance criteria.

- A scheme that introduces prototyping and review cycles early on in the DSS development process. It allows recursive and continuous review of the preliminary implementations against their own requirements, and thus helps the developers achieve the expected level of quality. AP is in a sense a 'watchdog' in ensuring that quality of the incidental implementations at the design phase meets the expectations.

- A procedure for exploring and acquiring views of various stakeholders and experts, and for assurance of consideration of both technology centered and user centered requirements. It brings into the DSS design process the expertise and experiences of the developers, as well as the expectations and knowledge of the users. In addition, reviewing the abstract prototypes by involving various stakeholders helps reconcile their requirements and at the same time brings in the objectiveness and independence from personal attitudes.

- A way to represent and store knowledge, which can be reused during the DSS development interval or in future DSS projects.

- A framework for gradual and systematic introduction of computer-based theories and methods into the industry. The idea behind is that involvement of future DSS users in the reviews of the abstract prototypes at the methods and pilot prototypes abstraction levels by default initiates and consolidates cooperation with the industry.

The main difference between AP and the traditional software testing methods is that AP does not assume the existence of executable codes or concrete DSS product in test floors. It tightens surveillance in the DSS development interval and: (i) helps to expose and avoid potential faults; and (ii) enables exploration of possibilities, choice of the best alternatives, and improvement of the selected alternative until all known requirements are addressed sufficiently. Its emphasis is on systematic review of the solution concepts, at various abstraction levels, and involvement of the DSS stakeholders in refining and in extending the initial concepts put forward by the developers. While the developers' or outside experts opinions help building the DSS according to the state-of-the art technological advances, the future users' views help the creation of the DSS that better support their works. The procedure sets out milestones and in-process deliverables in the design process, helps the developers define requirements for the abstract prototypes, and enable them to detect faults. It generally serves as a tool for ensuring that quality of the eventual DSS live up to the expectations. Clear understanding of the users as well as the infrastructure requirements, and early detection of the weak spots, increase the chance of creation of high quality and trustworthy DSS, and can possibly save a great deal of time, effort, and money.
Abstract Prototyping Software

A pilot software system, which supports the developers of DSS in performing AP activities has also been developed and tested as shown in Chapters 5 and 6. It provides software tools, which assist in the acquisition of requirements, authoring of the representations of the abstract prototypes, and also support the process of acquisition and analysis of opinions of the DSS stakeholders. Computer application primarily bring into AP the power to communicate, store and process information. The essence of AP being aided by computers is the fusion created by the application of its underlying schemes and the strength of computers in providing assistance to the developers or in enhancing their capabilities. The intention has been to make the AP procedure highly knowledge intensive by including requirements as well as information about various problems and their solutions in the AP software's databases. This makes the AP software more effective in supporting the DSS design process. It has been demonstrated during the application case studies that the AP software:

- Reduces the efforts required in the acquisition of requirements, fastens the requirement gathering process, and improves the consistency of requirements for the abstract prototypes. In addition, the AP software also helps: (i) keeping the requirements handy; (ii) avoiding the danger of deterioration of the understanding of requirements and the problem; and (iii) reducing the risk of overlooking requirements when designing DSS. Complete and consistent requirements in the AP context imply deeper reviews, fewer mistakes, clarity of what to implement, and better quality. Furthermore, good and exhaustive requirements at a given abstraction level imply fewer surprises in the subsequent levels, and signal greater chance of acceptance of the envisaged DSS.

- Provides powerful means for handling information about the problems and their solutions. The abstract prototypes that pop up at various stages of the design process can be represented in various forms, stored electronically, and reused.

Practical Applications

Practical application case studies have vindicated the AP concept as a sound quality assurance strategy. Significant increases in the number of requirements and the rate of acquisition of requirements have been witnessed as the result of the application of the AP methodology. It has also been demonstrated that AP helps the developers effectively identify weak spots in the early abstract prototypes, namely theories, methods, and algorithms, without the necessity of writing codes. Flaws in the proof-of-concept pilot prototypes can also be detected and eliminated without investing excessive efforts in coding. The application case studies have also attested that AP is a healthy strategy for shaping and directing activities at the design phase of the DSS development process. Based on the results of the application case studies, it is tempting to suggest that the DSS development community can benefit enormously from quartet prototyping and reviews of theories, methods, algorithms and pilot prototypes. However, it is important to mention that the AP concept has in this work been validated in a limited, controlled
research environment due to time and capacity limitations. Many more case studies need to be conducted to verify its validity as a quality assurance strategy.

Observations and Concluding Remarks

The goal of the research work reported in this thesis was to develop a generic framework for quality assurance of DSS in the early development stages. As a result, the abstract prototyping concept has been created. Apart from focusing on testing of the DSS before implementation (that is, at the design phase, prior to coding), it can also be described as a model for systemizing the disorganized design activities. It is a requirements-driven prototyping and review concept, in which the needs of the stakeholders of DSS, as well as technological constraints, provide the baseline for evaluation of the intermediate implementations of DSS at the design interval. It has been shown that it helps ensure that requirements are sufficiently being taken into the consideration at the design phase. In the framework of this concept, reviews are interwoven into the DSS design process. The concept assumes an iterative and incremental approach in which the envisaged DSS is repeatedly designed, prototyped and reviewed at different abstraction levels. This not only locates faults in the abstract prototypes, but also gives some level of confidence that the DSS development process is on the right track. The AP concept is novel in many respects. However, it is also indisputable that it embodies some elements of the existing software development, verification and validation concepts. What is perhaps most different is the structured prototyping and review of the abstract prototypes, and systematic involvement of various DSS stakeholders in this.

The most striking aspects of abstract prototyping is that it provides: (a) a theoretical framework, (b) a methodology, and (c) a suite of software tools, which support the developers in detecting and eliminating flaws prior to coding. It equips the developers with quasi-automated procedures for: (i) acquiring not only the requirements for the final DSS, but also for the intermediate products; (ii) sorting and reusing requirements; (iii) reusing the abstract prototypes; (iv) communicating and working in geographically distributed locations; and (v) collecting and analyzing views of various DSS stakeholders. The AP software supports handling of process information, seeking patterns of solutions and requirements, searching and exploration of contexts of solutions, and presentation of the analysis results. AP leverages intellectual assets of the developers by availing them with information, stimulates exploration, and generally serves as a DSS design model. Case studies have shown that the application of this technique can ultimately lead to realization of low-risk DSS since it provides means to constantly monitor and ensure that requirements for the final DSS product as well as for its in-process products are being taken on board harmoniously, at various abstraction levels. Also, by working in the framework of the AP concept, wrong decisions during the DSS design and the possibility of rework can significantly be reduced. However, it is important to emphasize that this practice of verification and validation at the pre-coding phases alone cannot guarantee quality of the eventual DSS. It would always be necessary to conduct classical software engineering tests after implementation.
The AP methodology has primarily been created for the DSS development projects, and perhaps it can be used in the development of other similar types of software, for instance, in the development of software tools that model physical phenomena. It is anticipated that with modest adjustments, it can also be used in a wide variety of development projects in software and web-based applications, for instance, as a software design model. Pragmatically speaking, in most cases it would not be required to utilize all AP tools. Instead, depending on the needs, type and/or the size of the project, some of the underlying methods or aspects would be utilized, ignored, or customized.

It is important to mention that some parts of this research, in particular the experimental part is by no means flawless. The application case studies dealt with only few DSS projects, and were at best capable of capturing only few perceptions. It is definitely desirable to conduct further case studies to explore more about the usefulness and the scope of the AP concept. It is also important to emphasize that this thesis neither marked or proposed the departure from the existing software development practices, nor argues that AP is a silver bullet procedure for quality assurance of the DSS. Rather, the AP concept is considered a companion of the existing software development strategies. It can be used within the frameworks of the traditional software development models or of the existing object oriented design methods as an additional quality assurance concept.

### 7.2 Contributions of the Research Work

The main achievements in this research can be summarized as follows:

- The processes of development of DSS have been analyzed and guises of the DSS at the design interval have been identified. These are theories, methods, algorithms, and pilot prototypes. It has been established during the application case studies that these are indeed the abstract prototypes, and they are the deliverables in the process of designing DSS.

- A state-of-the-art review of concepts, models and V&V techniques used in software engineering have been conducted and gaps (as far as the DSS development and testing is concerned) diagnosed\(^3\).

- A theory and a formal methodology for pre-implementation testing of DSS have been created, and arsenals of prototype software tools, which support various pre-implementation testing activities, have been developed and tested.

- It has been testified through the application case studies that AP:
  - Offers a sound model for shaping the DSS design activities.

---

\(^3\) Refer to Chapter 2 for the complete review and the findings. In short, the review revealed that the existing software engineering strategies are to a large extent either too local, partial, or too general, and do not scale to precisely match the needs in the development of DSS. Activities in the phases of the traditional software process models, in particular the design phase are coarsely defined and do not conform to the actual demands of the DSS design process.
Enlarges the scope of requirements specification, and provides a framework for handling requirements for the abstract prototypes.

Provides a sound and an objective\textsuperscript{4} procedure for detecting flaws in theories, methods, algorithms, and in codes for pilot prototypes (by involving various DSS stakeholders early on), which otherwise may or may not be discovered during testing and debugging phases.

7.3 Limitations to Problem Solution

Chapter 6 and Appendix E have shown how beneficial AP can be. However, there are some issues worth considering in future research. One fundamental issue is how generic the AP methodology can be. It is still difficult to say anything about this, because the methodology has not yet been applied in many practical situations. More case studies need to be conducted in a variety of DSS development projects. Moreover, nothing, as yet, can also be said about the influence of the AP methodology on the productivity of the DSS development process. This requires more studies as well. There are indications in the literature that the software developers' community agrees that using systematic development and testing methods is necessary, but some are skeptical, and argue that some methods slow down the development process and limit creativity. Apparently, time consumption and the economics of using formal methods are typically uncertain, and in general terms formal methods poses constrains, and as a consequence tends to limit creativity. Certainly there are open questions on how to maximize the effectiveness of the AP methodology.

Furthermore, there are some research issues in software engineering that remain unresolved and thus affect AP as well. For example, it is hard for the developers to satisfy themselves that all requirements have been taken into consideration. There is also the question of validity of the AP metrics and the measurement method used. This thesis does not claim the completeness or reliability of the metrics and the measurement method adopted. This has not been among the core directions of this research. The method for measuring the acceptability of the abstract prototypes and levels of fulfillment of requirements have been developed for use in the proof-of-concept application case studies exclusively, and have been tested in a limited and controlled research environment. In addition, the decisions on the limiting values such as the ceiling value of the \( \alpha \) index for an abstract prototype to be considered as unacceptable, or a threshold value of \( \Phi \) index for a requirement to be regarded as fulfilled to a satisfactory level remain on the hands of the developers. This retains the elements of subjectivity in AP. Thus, it is important to emphasize that the proposed metrics only give approximate quantitative indications, and are not necessarily comprehensive enough, or readily transferable to other projects or firms.

\textsuperscript{4} This is according to the opinion of those who applied or worked in the framework of the AP concept. Refer to Chapter 6 and Appendix E.
7.4 Recommendations for Future Work

Despite the contributions of this work, still there are open research issues that must be addressed in order to achieve the goal of having a more effective methodology for quality assurance of DSS in the early development stages, or extending the usability of the achieved results. To enhance the AP concept and to widen its application domain, further research needs to be carried out in various directions, for instance, to:

- Explore how the AP concept can best be used alongside the major traditional software process management strategies.

- Investigate how the AP concept can effectively be used in conjunction with conventional software V&V strategies.

- Study and establish the consequences of the application of AP on time to market and development costs.

- Investigate the possibility of extending the application domain of the AP concept to include other similar software products.

- Extend the scope of AP beyond software products to include, for instance, pre-implementation testing of artifactual products.

In addition, further research is required to verify whether using the AP methodology enlarges the scope of requirements specification, and whether this results into the improvement of quality of the DSS.
References

Own Publications


References

Design", Proceedings of the 1999 International CIRP Design Seminar, Enschede, The Nether-
lands, March 24-26, Kluwer Academic Pub-
lishers, pp. 415-424.

Other Publications

Agarwal, R., De, P., Sinha, A. P., Taniru, M., (2000) "On the Usability of OO Representa-

Altov, H., (1994), And Suddenly the Inventor Ap-
peared, Worcester, MA: Technical Innovation
Center.

Altshuller, G. S., (1984), Creativity as an Exact
Science: The Theory of the Solution of Invent-
tive Problems, New York: Gordon and Breach.

Armour, P. G., (2000) "Five Orders of Ignorance",

Armour, P. G., (2001 a) "The Laws of Software

Armour, P. G., (2001 b) "Software as a Currency",

Armour, P. G., (2001 c) "Matching Process to
Types of Teams", Communications of the ACM,

Armour, P. G., (2001 d) "Zeppelins and Jet Planes:
A Metaphor for Modern Software Projects",

Avgerou, C., Cornford, T., (1993), Developing
Information Systems – Concepts, Issues and
Practice, Macmillan, Hong Kong.

Bashir, I., Goel, A. L., (2000) "Testing of Object-
Oriented Software: Life Cycle", Quality Tech-
niques Newsletter, Software Research Inc.,

Project: Towards Improvement-Oriented Soft-

Bayer, H., Holtzblatt, K., (1998), Contextual De-
sign: Defining Customer Centered Systems,
Morgan Kaufmann Publishers, Inc.

Beizer, B., (1999) "Testing: Best and Worst Prac-
tices", Testing Techniques Newsletter (TTN)

Beizer, B., (2001 a) "Software Is Different", Qual-
ity Techniques Newsletter, Software Research

Beizer, B., (2001 b) "Test Automation Issues",
Quality Techniques Newsletter, Software Re-
search Inc., http://www.soft.com/News/QTN-
Online, December Issue.

Berard, E. V., (1993), Essays on Object-Oriented

Berezins, V., (2000) "Software Prototyping", Ency-
clopedia of Computer Science, Fourth Edition,

Birkhofer, H., Sauer, T., Berger, B., Kloberdanz,
to Get Them to Work ", Proceedings of the 3rd
Engineering Design in Integrated Product De-
velopment (EDIProd 2002) Seminar and Work-
shop, Lagow, Poland, pp. 29-36.

Bloor, M. S., Owen, J., (1995), Product Data Ex-
change, UCL Press, London, UK.

Blumenthal, S. C., (1969), Management of Infor-
mation Systems: A Framework for Planning and
Development, Prentice-Hall.

Boehm, B. W., (1988) "A Spiral Model of Software
Development and Enhancement", Computer,
No. 31. Vol. 5 pp. 61-72.

Boehm, B., Port, D., Al-Said, M., (2000) "Avoid-
ing the Software Model-Clash SpiderWeb",

Bolton, M., (1999) "Effective Beta Testing", Test-
ing Techniques Newsletter (TTN) Software Re-
search Inc., http://www.soft.com/News/TTN-
Online, October Issue.

Booch, G., (1991), Object Oriented Design with
Applications, Redwood City, CA: Benjamin
Cummings, pp. 39.

Booch, G., (2001) "Developing the Future", Com-
munications of the ACM, Vol. 44 No. 3. pp. 119-121.

Broek, J. J., Horváth, I., Smit, A. de, Lennings,
Shape Decomposition for Thick Layered Object
Manufacturing of Large Sized Prototypes" in
Revue Internationale de CFAO et dlnformatique
Graphique, edition.


Deutsch, M. S., (1982), Software Verification and Validation, Prentice-Hall Englewood Cliffs, NJ.


References

national Conference on Engineering Design, August 24-26, Munich, Germany, pp. 1525 - 1530.


Appendix A: Abstract Prototyping Software

This appendix provides further description of software tools built to support the developers of DSS in performing AP activities. A higher-level architectural design of the AP software and the implementation details are presented first. The need for computer support, tasks supported and the extent of support provided, as well as the scope of the AP software are also addressed in a question and answer (Q&A) style. A gallery of snapshots of various GUIs is provided to give impression on what the prototype AP system offers.

A.1 Introduction

The AP system [Figure A.1] is basically an information system. It consists of hardware, an operating system, application software, a database, and a knowledgebase. Some of the AP operations are automated while others are not. Performance of the AP system is a combination of the performances of its elements, namely the operating system, the application software, and of the hardware equipment.

A.2 Basic Application Software

The AP application software is made up of a collection of procedures to help the developers model, edit, process, store, retrieve, sort, distribute and display various kinds of AP information presented in various different forms. As mentioned in Chapter 5, the application software has largely been built using Microsoft technology. Some existing commercial software tools, namely (i) Microsoft Equation Editor, (ii) Microsoft Drawings Editor, (iii) Microsoft PowerPoint, and (iv) Microsoft Word Processor have been summoned to be used outside their mainstream domains to provide the AP application software with means for creating textual and graphical representations of the abstract prototypes. Excel has been adopted as a statistical analysis engine of the prototype AP application software. Some of its utilities have been customized to provide means for analysis of information and presentation of results. Excel is also the platform for implementation of the opinions gathering module. CLIPS has been used as a language for development of the advisory module, one of the auxiliary utilities of the AP application software. The real value here is that capabilities and features of several existing software systems are made available to support AP in a seamless user-friendly environment. Several other utilities have also been implemented from scratch using various different high level programming languages such as Visual Basic, Visual Basic for Applications, HTML and Visual C++.

A.3 Warehouse: The AP System's Database and Knowledgebase

The database and the knowledgebase constitute the AP software's warehouse. Data stored in and maintained by the AP software is called the database. Of course such data is stored on auxiliary memory devices such as discs or tapes. On the other hand, the facts (or knowledge) stored in and maintained by the system constitute the knowledgebase.

As mention in Chapter 5, there are three different databases for storing electronic representations of the abstract prototypes, requirements and AP experiment histories. Textual, graphical, audio, or video descriptive data, as well as files of various kinds, including executables can be stored. The database has been implemented using Access, a relational database management system (RDBMS). The real value here is that capabilities and features in Access such as for entering, editing, validating data in tables using queries, and Internet-related features for creating HTML documents are available to support information representation and processing during AP.
Various manipulation algorithms have been implemented to provide users with means for finding various kinds of data that may be required. And yet users can still create mechanisms for searching any kind of requirement or abstract prototype information that they may wish to have from the system's database. In its current form, two kinds of information are dealt with, namely information about testing techniques in general, and information related to the AP concept [see Figure A.12].

A.4 User Interface

Various utilities that constitute the AP application software have been integrated and appear as a singular system, with one common user interface. Every utility of the AP software can be operated via this user interface. Even access to the database and the statistical software is masked behind the AP system's UI, which provides an effective interface for efficient and valid information access. The user interface source code is written in Visual Basic and Visual C++.

A.5 Hardware, Operating System, and Associated Software

The hardware [the machines, including the central processing units (CPU), servers and various input and output devices] and the system software [also commonly known as operating system] constitute the platform. The prototype AP software runs on Windows 98/NT/2000/XP. The associated software products that the AP application software exploits, and must be installed alongside the basic software are: (i) Microsoft Office 97 or higher, and (ii) CLIPS.

A.6 Non Computerized Operations

In AP, automated operations are usually supplemented by manual operations. The AP application software accepts, stores, processes, sort and displays strings of symbols such as digits, alphabetic characters and special symbols grouped in various different ways. Users of the AP software ascribe some value or meaning to the displayed information. Almost all critical decisions such as selection of requirements to take on board in AP or kinds of subjects to incorporate rest upon the users of the AP software: the DSS developers.

A.7 Answers to Questions Frequently Encountered when Using the AP Software

Users of the AP software frequently asked some questions, for instance, on capability of the AP software, its limitations, and so forth. Most probably the readers of this thesis might have the same questions. This section, therefore, presents the frequently asked questions (FAQs) as well as answers to the questions.

A.7.1 Why need the AP software?

If a software development team or company is involved in a DSS project and is concerned about quality and trustworthiness of the eventual DSS product, then the AP software can provide a helping hand in assuring this. Also, if a DSS development team or company want to have confidence that the development process progresses on the right track, the abstract prototypes have the desired quality, and that the eventual users would accept the envisaged DSS, then the AP procedure can be very helpful. Symbiotic application of the AP methodology and software helps developers achieve these goals.

A.7.2 Which unique tasks are supported?

AP tests focus directly on quality assurance at the DSS design phase, and the end results are more consistent and complete abstract prototypes. The AP methodology [see Chapter 4] defines the necessary pre-implementation testing operations. Based on its underlying schemes, tests on the early implementations can be run more systematically and efficiently compared to ad hoc manual procedures. The AP software has been created to support quasi-automated processes such as (i) acquisition of requirements, (ii) representation of the abstract prototypes, (iii) gathering of opinions, and (iv) analysis of information
A.7.3 What kinds of tests can the AP procedure help?

The developers can test the compliance and assure quality of the abstract prototypes with or without involving the DSS stakeholders or other experts. Also, the review criteria can be used as a simple checklist for counterchecking compliance of various abstract prototypes to incidental requirements.

A.7.4 How serious are the defects that AP can catch?

Flaws in the DSS can result in very serious consequences indeed. Consider, for example, a DSS used in the design of automobile or its parts, and suppose its algorithms are flawed, and say, for instance, truncation errors accumulate and this goes unnoticed. If such a DSS is put in service, this can lead to creation of inaccurate designs; which can ultimately endanger lives of commuters, pedestrians, and so forth. Imagine also, for instance, a utility for handling simple modifications in a conceptual shape design software. If the inherent methods are too rigid or software tools are not developed according to the ways that better suit the designers, for instance, demanding them to have knowledge of the mathematics or physical phenomena behind the application, then it can be difficult for the designers to accept them. Even if adopted contrary to wishes of the designers, the resulting DSS would be hindering rather than facilitating the shape conceptualization process. Ultimately the company using such software would most probably be losing a lot of revenue. These kinds of defects are serious, and AP can offer a tremendous advantage by helping detection of such problems virtually possible early on.

A.7.5 What it takes to use the AP software?

As can be seen from its design [in Chapter 5], the AP software is a general-purpose tool that can either be used as it is, or customized to suit any DSS project. Data in the database can be updated (added or removed) as required, and even the database schema can be modified and scaled to particular DSS projects. A different database can also be created altogether and used by the AP application software.

A.7.6 How does the AP software improve quality of the DSS?

The AP software system is equipped with a searchable database, which among other information contains sorted requirements, which can be reused during the development process. One of the common problems in software development is that there is always a good chance of neglecting substantial requirements unintentionally. As can be seen from the application case studies [see Chapter 6], with the support of the AP software, many requirements can be taken into the consideration. This improves the chance of developing quality DSS.

A.7.7 How does the AP software speed up the DSS development process?

The AP software speed up the AP process in two ways, namely: (i) by reducing the requirements acquisition time [refer to Chapter 6], and (ii) through the reuse of the abstract prototypes. Normally, requirements gathering is a lengthy process, and extends to, for instance, holding series of brainstorming meetings and carrying out needs analysis. With the AP software, the requirements are stored in the database and a search mechanism has been developed with which the developers can find the requirements that match various problems. Another search mechanism has been developed with which the developers can find the abstract prototypes that match various problems. This means that instead of starting from scratch, the developers can reuse the existing implementations. The advantage here is that if there is a matching abstract prototype, the development work can be reduced to fine tuning the existing implementation. In this way, the time to market for a DSS can also be reduced significantly.
A.7.8 Does the AP software automate the DSS development process?

The goal of the AP software is not to achieve full automation of the AP process, but rather to provide software tools to assist the developers in AP. Computer support of AP is important in many respects. For example, computers enable massive amounts of information about the abstract prototypes to be stored and in turn processed and availed to the AP subjects or the developers even at distant geographic locations, allows for unlimited number of requirements to be stored, and enables the computations required during the information analysis stage to be performed faster and accurately. The role of humans in AP is also vital. Human beings make major decisions.

A.7.9 What are the other benefits of the AP software?

Apart from contributing to quality improvement, the AP software eradicates the common bizarre ritual of spending valuable resources and time in the development of various abstract prototypes and throwing them out at the end of the day. This is due to the fact that it facilitates the reuse of the abstract prototypes. The abstract prototypes in various forms of representations can be stored, and a search mechanism has been developed with which the developers can find the abstract prototypes that match various problems.

A.8 A Gallery of Snapshots of the AP Software's GUI

This section presents a gallery of snapshots of various user interfaces of the AP software. The goal is to illustrate how the AP software can support the developers of the DSS in doing the AP activities in actual practice.

![Figure A.2: Requirements input/editing session](image-url)
Figure A.3: Finding existing solutions: Previous representations retrieved via the GUI of the AP software.

Figure A.4: Authoring sessions: Representing a theory descriptively using Microsoft's equation editor; an algorithm graphically using Microsoft's PowerPoint launched via the GUI of the AP software.
Figure A.5: GUI of the database

Figure A.6: Requirements processing session: Using the MDRS utility to find a provisional list of requirements for an AP review from the database
Figure A.7: Typical AP information analysis session: Determination of the acceptability index for an abstract prototype

Figure A.8: Abstract prototypes information input/editing session
Figure A.9: Matching problem to requirements: An appropriate requirements model retrieved via the GUI of the AP software.

Figure A.10: Typical requirements ranking session using a customized Microsoft's Excel utility run from the GUI of the prototype AP software.
Figure A.11: Investigation of improvement strategy: Requirements ranked according to the collated $\rho$ values, and then collated $\phi$ values using a customized Excel utility operated via the GUI of the prototype AP software.

Figure A.12: Knowledge handling utilities: An expert system and a hypertext utility - handle information about quality assurance techniques.
Figure A.13: Web browser: Providing subjects and developers with unimpeded access to requirements and representations of the abstract prototypes published on the web.

Figure A.14: On-line mentor: A hypertext tool providing expert guidance in the selection of: (i) forms of representation of the abstract prototypes, and (ii) subjects.
Figure A.15: Web based file-uploading utility: Used by subject in remote locations to submit completed information gathering matrices.

Figure A.16: Finding requirements from a common database via the Internet.
Figure A.17: Typical electronic information gathering matrix with macros

Figure A.18: Typical session on the investigation of the implications of choosing a solution
Figure A.19: Using an ODBC driver to connect to and to edit fields of the requirements database

Figure A.20: Using an ODBC driver as a connection to the requirements database to view its contents
Figure A.21: Using the ODBC driver to connect to and to manipulate the 'analysis results' database

Figure A.22: Pre-prepared electronic directions - must be recalibrated to suit specific needs
Appendix B: Protocols for Acquisition and Management of Requirements

It is universally acknowledged that bad requirements are the key cause of most software projects failure, including failure of DSS projects; and that acquiring and using good requirements is central to success of any software project. In AP, the level-wise requirements carry the vision for the abstract prototypes and are vehicles for focusing the DSS design process. Requirements specification is also crucial in managing customers' expectations, and should primarily be a way of communication between the developers and the users of DSS. A fair proportion of the development efforts must therefore be invested to ensure collection of good requirements. In this appendix, some tips and tricks on how to write good requirements for AP purposes are presented.

B.1 Tips

The following are some precautions to take into account during formulation and handling of requirements for abstract prototypes or DSS. In fact, most of these tips are also valid in traditional requirements engineering and software development processes in general.

B.1.1 Use of Words

Different words can be used to send different messages. For instance, the word 'shall' may be used to indicate mandatory requirements and 'will' to indicate guidance or non-essential functionality. Other words such as may, could, can and should, can also be used. Such words may be used as keywords in making various kinds of search, for instance, in the investigation of to what extent individual requirements should be treated, and so forth.

B.1.2 Characteristics of Good Requirements

The following are some tips on the characteristics that a requirement statement or a complete requirements model must possess:

Clarity: The goal of a requirement should be obvious and free from wordiness. The reader shouldn't have to think too hard about the meaning of the requirement.

Completeness: Every part of the system envisaged must be described in the requirements model. The requirements should be free from obvious omissions.

Consistency: A requirement should be free from unnecessary jargon, homonyms and synonyms, and should make sense in its context. It should be worded within documentation standards or conventions set out. Conflicts or contradiction with any other requirements should be avoided as much as possible.

Correctness: A requirement should contain precise information.

Details: Excessive details should be avoided. A requirement should not infringe, for instance, on the design process by describing how things will be done, rather than what should be done.

Identifier: Requirements should contain keywords, which provide visibility to the requirements. For example, the word 'shall' can be used to imply mandatory requirements while 'will' can imply guidance.

Implications: A requirement should avoid implying that something extra would be allowed.

Reasonable: The targets implicated by a requirement must be justifiable and achievable within the constraints of project's budget, schedule and available resources.
Relevant: A requirement should be expressed either as mandatory, a guidance, or as a non-essential functionality. It shouldn’t be merely narrative with no bearing on the functionality of the system.

Testability: Vagueness in requirements must be avoided. A requirement must be very specific and quantifiable. Tolerances and ranges should be expressed, there should be a reasonable way of testing them, and the effects of the requirement must be visible.

Traceability: Sources of requirements should be traceable.

Unambiguous: A requirement should be interpretable in one way only.

Unique: A requirement should uniquely be the only one in the requirements model that implicates the same aspect.

Usability: The success of a requirement obviously comes from its use. Thus, a requirement must be valid to all stakeholders that rely on its content.

Verifiable: A requirement must be verifiable, and every aspect must be precise.

B.1.3 What to Observe when Formulating Requirements

The following should be observed when formulating requirements for AP:

- The reason(s) for existence and use of a requirement should be clear.
- Assumptions must be stated beforehand. This reduces possibility of misinterpretation. An unstated element in a requirement can take on different meaning, and this can possibly leads to incorrect implementation or disputes in various development phases.
- Cross-referencing must be avoided as much as possible. Cross-references, for example that merely point to a standard, or another document, imply that the whole thing applies. Ultimately, this can lead to unexpected blowouts in scope.
- Dependencies of a requirement, for instance, on other requirements, external software, projects or technologies should be identified.
- A requirement must be uniquely identified. The simplest and best way to do this is to number the requirements, say, from 1 to n.
- A requirements document should be organized in such a way that a change to an individual requirement does not cause excessive impact on other items in the document.
- The importance of a requirement should be obvious, and this can be implied through wording of the requirement. Apparently some requirements are more important than others, and this must be exhibited boldly.
- The abstraction level(s) at which a requirement can be tested should be specified. Actually, some requirements can be tested in one or more than one abstraction level, while some requirements cannot be tested at all within the AP interval.

B.2 Management of Requirements

The following are some guidance on management of requirements for AP of DSS.

- As soon as the identifiers are assigned to the requirements, never change them, or recycle the identifiers from deleted requirements. This can cause confusions.
- Avoid deleting any requirement from the requirements specifications model. If any requirement is not being used, then the best way is to strike it out or hide the text. In this way, it could be easier to recover it if needed.
Appendix B: Protocols for Acquisition and Management of Requirements

- Don't be tempted to use automatic identifiers assigning tools in place of manual assignment of identifiers. There is considerable risk that the requirements document will look different from site to site or even machine-to-machine. One automated change in the document can result in unintentional requirement re-identification and a traceability nightmare.

- Make a cautious judgment on automation. There is always a danger of removing essential words in requirements, for instance, when cutting or pasting texts.

- Create and maintain a DSS project's glossary of terms. Whenever there is even a slightest change of jargon, a new word, or an acronym used outside its normal meaning, it should be appropriately defined and included in the DSS project's glossary of terms.

B.3 Requirements Used in the Application Case Studies

As mentioned in Chapter 6 and elaborated in Appendix E, several case studies have been conducted. One of the primary activities in these case studies had been acquisition of the 'level-wise' requirements, putting them in appropriate contexts, and eventually rephrasing them into a language that can easily be understood by the targeted subjects. The developers closely kept an eye on the above-mentioned tricks and tips when formulating or handling requirements.
Appendix C: Glossary of Terms

This appendix presents definitions of the words or phrases adopted and used somehow differently from their common context in this work to describe the AP concept.

**Abstract prototype:** An intermediate implementation of a design support software at the design phase. Can be a theory, a method, an algorithm, or a pilot prototype. The term *abstract prototype* is used synonymously with *in-process implementation*.

**Abstract prototyping:** A computer supported methodology for verification and validation of the abstract prototypes against the *review criteria* by systematically involving various stakeholders.

**Abstraction:** The act or action of abstracting. Denotes extraction of essential details about an item or group of items, while ignoring inessential details. Allow people to deal with concepts apart from particular instances of those concepts.

**Abstraction level:** Refers to the level of completeness of an abstract prototype relative to the final DSS. Reflect gradual transformation and reappearances of a DSS in various contexts.

**Acceptability index:** An index that give an idea about the extent to which the abstract prototypes meet user and technological requirements.

**Algorithm:** A sequence of instructions used in a fixed order to find an answer to a question. Provides details of the flows and actions in the envisaged DSS. Gives insight into issues such as how long it will take to prepare working codes, what kinds of resources will be required, what will be the capability of the imagined system, and what will be the performance characteristics.

**Algorithms level:** The third abstraction level of AP. In this level, details of all features of a DSS including its user interfaces, knowledgebase and database management system are concretely specified. AP in this level ensures that the desirable algorithmic characteristics that eventually contribute towards guaranteeing the acceptance of the DSS are taken on board.

**Design support software:** Refers to any software used in industrial or mechanical engineering design, variously known as CAD, CAE, or CAD/CAM software systems.

**Information gathering matrix:** A specially formatted paper-based or electronic tool used in the collection of opinions of subjects during AP.

**In-process implementation:** Output of a particular DSS design task. In the design phase, an in-process implementation can be a theory, a method, an algorithm, or a pilot prototype. The term *in-process implementation* is used synonymously with *abstract prototype*.

**Level of implementation:** A distinct intermediate step in the development of a DSS at the design stage. The four *levels of implementation* in AP are (i) theories, (ii) methods, (iii) algorithms, and (iv) pilot prototypes levels. The phrase *level of implementation* is used synonymously with *stage of the DSS design process*.
Level-wise requirements: Requirements for an abstract prototype. Specifies the desired characteristics of the individual abstract prototypes rather than of the entire DSS system.

Method: A functional representation of the solution concept, with the consideration of the appropriate sequence of theories and organization of parameters from computation point of view, but without formal structural details or characterization of the solution into a form interpretable by the device to be used for solving the problem. Expresses the way or manner of doing things.

Methods level: The second abstraction level of AP. Concerns with how theories can be put in use. The focus at the methods level is on sequencing of the theories, with the view of attaining optimal flow of processes. The aim of methods level AP is to ensure, prior to coding that usability problems do not get designed into the DSS.

Pilot prototype: An executable core function of the anticipated DSS (i.e. limited implementation of the DSS, or in other words an implementation of some selected vital algorithms of a DSS).

Pilot prototypes level: The fourth abstraction level of AP. AP at this level involves reviewing the solution concepts by running a pilot prototype.

Representation of the abstract prototype: A description of the DSS. Portrays certain quality characteristics of the imagined software, e.g. the functionality or appearance of the envisaged system. Does not represent the envisaged DSS as precisely and completely, as it would be seen. Provides means to study the behaviour of the whole software or key parts of the software. Helps to invoke ideas on how the solution to the problem will be and offers concrete basis for discussing difficulties, clarifying problems or preparing discussions between the developers and future users or the management.

Requirement statement: A clause declaring a need, a demand or an order for a specific characteristic feature of either an elementary part or the whole DSS system. Specify qualities that a DSS or its components must possess in order to attain compliance or solve a problem. Indicates what the DSS system will do.

Requirements model: A list or document of requirements describing quality characteristics for an abstract prototype and providing additional connectivity information about the individual requirements. Comprehensively describe quality characteristics of the DSS at the respective design stage.

Review criterion: A statement that carries the semantics of a user or a technical requirement. Serves as metric for assessment of the completeness and consistency of an abstract prototype.

Review: Determination of the conformance of an abstract prototype to review criteria.

Specification: A 'version' of requirements written for software development purposes. Have the same information as the requirements, but couched in a formal technical language rather than natural language, using formal software engineering terminology.

Stage of DSS design process: A definite stage when designing DSS. The DSS design stages include creation or selection of theories, formulation of methods, development of algorithms, and writing codes for pilot prototypes.
Subject
An individual involved in the review of an abstract prototype. Voices his/her opinions by completing an information-gathering matrix.

Theories level:
The first abstraction level of AP. In this level, theories are optimised based on knowledge of the problem and of the solution constraint(s). Descriptions of theories in text, graphical, audio or video form can be used as a basis for forming opinions on the appropriateness of theories.

Theory:
A general principle for a specific part of the subject, in form of a statement or group of statements, equations, or in any other form of representation, established by reasoned ideas or arguments, based on known facts. Intended to explain a particular fact or event, explanation for which certain proof may still be needed, but which appears to be reasonable.

Warehouse:
Knowledge or data repository. Can be a database or a knowledgebase.

Weighting factor:
A value assigned to a quality characteristic or a review criterion by the developers, taking into the consideration views of the subjects. Indicates the significance of a review criterion on the acceptance of an abstract prototype.
Appendix D: Metrics and Measurements in Abstract Prototyping

This appendix presents the metrics and measurements used in measuring how well the abstract prototypes are being developed. The information-gathering tool and tips for selection of subjects are presented first. Then, the types of values that can be used in the assessment of the acceptance of the abstract prototypes are described. Finally, a procedure used in the analysis of the collected data is presented.

D.1 Background

Metrics and measurements make it possible for the abstract prototypes to be analyzed quantitatively. They enable the developers of the DSS to evaluate the acceptability and quality of the abstract prototypes. Basically, requirements are criteria for measuring the acceptability. A numerical value assigned against a review criterion by a subject representing a given stakeholder community constitutes the measure of its fulfillment. A specially designed form, enlisting all review criteria and comprising provisions for the subjects to assign values is used to gather opinions of the DSS stakeholders.

D.2 Opinions Gathering Tool

A specifically forged tool used in collecting views of the DSS stakeholders in AP is referred to in this thesis as information-gathering matrix [Figure D.1]. This in a way is a dedicated decision matrix [refer to e.g. Steward, 1981], which incorporates intuitive provisions for assigning numerical values, upon which fulfillment of requirements and acceptability of the abstract prototypes can be judged. Subjects are required to input into the information-gathering matrix values that express their opinions as well as competencies. There are provisions for indicating the levels of fulfillment of the review criteria (ϕ), relevance of the review criteria (ρ), as well as their confidences (χ) with regard to the assigned values. This form can be presented to subjects electronically or in paper form, depending on the wishes of the individual subjects. Views of the DSS stakeholders can be sought at four abstraction levels [Figure D.2], and at each level, only the relevant criteria are used.

![Information-gathering matrix](image)

Figure D.1: Information-gathering matrix
D.3 Selection of Subjects

Success of AP depends on, among other things, the composition of the panel that reviews the abstract prototypes. It is universally understood that quality preferences vary among stakeholders. Different stakeholders typically have different functional and non-functional requirements. Typical DSS development projects have various stakeholders, and as stated in Section 3.5, there are two broad classes of stakeholders in AP, namely (i) users, and (ii) developers (experts). The kinds of users to incorporate in AP as subjects vary according to the task that the envisioned DSS supports. For example, for an engineering analysis DSS, the users may include design engineers, and other indirect users such as drafters and process planners. On the other hand, expertise required in the development of DSS can be available within and from outside the development team. Consequently, experts to engage as subjects may be from within and/or outside the development team. A successful AP exercise must take on board the representatives of various stakeholders. Incorporation of a wide spectrum of stakeholders is an indirect way of counterchecking whether their requirements are being taken into the consideration.

D.4 Types of Values

The approach adopted in gathering opinions about quality of the abstract prototypes at the design phase is based on point allocations [see for example, in Matousek, (1963); McDonagh-Philp, (1999)]. Depending on the nature of the review criteria, the following sorts of values may be used when completing an information-gathering matrix:

- **Binary values:** These apply to review criteria that require simple "Yes" or "No" value. For example, asking whether a solid modeling software allows its users to add their own personal library leads to a response whose value is simply "Yes" or "No". Practically this means specifying maximum or minimum value.
• **Comparative values**: These values are for review criteria that require relative values. For example, 'similarity' can be an important factor in the acceptance of a new piece of software, say, when developing a functionality that resembles one that has been successful in a different profession (for instance design vs. medical). If the subject can guess how a particular function works, then s/he will be happy with the software. It is hard to imagine a measure for similarity other than a simple relative measure, which awards a score on a rating scale.

• **Numerical values**: A response can also be in form of a metric value. This is typical, for example, when asking subjects about specific numerical values, for instance, how many personal libraries can be defined.

Before subjects can assign values against the review criteria, they need to know facts, which can be discovered by referring to representations of abstract prototypes presented in the form of text, audio or video. Some sorts of test may also be carried out to widen their knowledge before they can decide on values to assign against the review criteria. Typically, this requires assignments to be set out [see Section E.1] and the review subjects to 'experiment' with the abstract prototypes prior to giving their opinion. Values assigned against the review criteria ultimately give rise to quantitative measures of the acceptability of the abstract prototypes and of fulfillment of requirements.

### D.5 Analysis

The information gathering procedure presented in section D.1 enables the developers to collect opinions, and an enormous amount of data is typically produced. Consequently, there are challenges of (i) how to analyse the collected data and present the analysis results, and (ii) how to utilise the analysis results to help decision-making during the DSS design process.

#### D.5.1 Analysis Procedure

Figure D.3 shows a tabular representation of the proposed analysis method. The responses ($\psi(x)$) are compiled first. Weighting factors ($\omega$) must be specified to provide some kind of moderation, since the influences of various requirements on quality of abstract prototypes are typically not equal. For the purpose of objectivity of the analysis process, $\omega$ values must be determined and fixed in advance, well before analysis.

Figure D.4 depicts the analysis process. Three measurements, namely; (i) the acceptability ($\alpha$), (ii) relevance ($P$) and (iii) confidence ($X$) indices are determined. They collectively serve as criteria when judging the appropriateness of the abstract prototypes. They offer quantitative criteria for assessing how well the abstract prototypes have been implemented and indicate how close the ideal solution has been approached. They also help in the selection of the best alternative(s) when several alternative solutions prevail.

<table>
<thead>
<tr>
<th>The evaluation criteria</th>
<th>Scores $\mu_i$ as specified by subjects</th>
<th>Score, $\sigma_i$</th>
<th>Weight</th>
<th>Weighted Scores, $\sigma_{ei}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>$\mu_{11}$ ... $\mu_{1m}$</td>
<td>$\sigma_1$</td>
<td>$w_1$</td>
<td>$\sigma_{e1}$</td>
</tr>
<tr>
<td>$R_2$</td>
<td>$\mu_{21}$ ... $\mu_{2m}$</td>
<td>$\sigma_2$</td>
<td>$w_2$</td>
<td>$\sigma_{e2}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$R_{n-1}$</td>
<td>$\mu_{n-11}$ ... $\mu_{n-1m}$</td>
<td>$\sigma_{n-1}$</td>
<td>$\omega_{n-1}$</td>
<td>$\sigma_{e(n-1)}$</td>
</tr>
<tr>
<td>$R_n$</td>
<td>$\mu_{n1}$ ... $\mu_{nm}$</td>
<td>$\sigma_n$</td>
<td>$\omega_n$</td>
<td>$\sigma_{en}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\sigma_f = \sum_{i=1}^{n} \sigma_{ei}$</td>
</tr>
</tbody>
</table>

**Figure D.3: Analysis matrix**
D.5.1.1 Acceptability Index

The acceptability index ($\alpha$) can be defined as a metric for measuring the extent to which an abstract prototype fulfills requirements and how it compares with the ideal solution. It therefore helps developers obtain feeling on how the solutions in hand differ from 'perfect' abstract prototype. The ideal solution corresponds to the ideal review situation in which all subjects allocate maximum score to all review criteria. A threshold acceptability index value ($\alpha_{th}$) can be set such that if the obtained value happens to be greater than or equal to $\alpha_{th}$ then the abstract prototype can be considered suitable, and work on the subsequent level can be started. When more than one solution exists, $\alpha$ for all alternatives must be determined. The best alternative is then one with highest value of $\alpha$. $\alpha$ is defined mathematically as follows.

$$\alpha = \frac{\omega_j \sum_{j=1}^{m} \mu_{ij}}{\varepsilon_j},$$

where; $\mu_{ij} = \phi_i \rho_j X_{ij}$.

It is important to underline that when an abstract prototype of the design phase is tested for compliance with the requirements, the results may not always be a clear pass or fail. Most often in-between results (that is, partial success or failure) are obtained. In this case, it would be required, for instance, to revisit the criteria used and the abstract prototype itself must be improved.

D.5.1.2 Relevance Index

The relevance index $P$ for a review criterion $j$ can be defined as the ratio of the sum of the assigned relevance values to the sum of the maximum possible relevance values, that is

$$P_j = \frac{\sum_{i=1}^{n} \rho_i}{n \rho_{max}} \quad (D.2)$$

where $n$ is the number of subjects involved in the review. This index indicates how relevant, in the opinions of the subjects, the review criterion and consequently the requirement is, with respect to the ongoing AP review. A higher value of $P$ implies that the review criterion (requirement) in question is legitimate.

D.5.1.3 Confidence Index

The confidence index ($X$) can be defined as the ratio of sum of the confidence values and the sum of the maximum possible confidence values, that is,

$$X = \frac{\sum_{j=1}^{m} \sum_{i=1}^{n} X_{ij}}{mnX_{max}} \quad (D.3)$$

where $n$ is the number of subjects involved in the review and $m$ is the number of review criteria used. This index indicates how confident the subjects have assigned $\phi$ and $\rho$ values, and shows how knowl-
Appendix D: Metrics and Measurements in AP  173

eadable and reliable they are. The higher the value of the confidence index, the more the knowledge-
able and reliable the subjects are. Thus, the confidence index in a way gives clue on the reliability of
the AP review.

D.5.1.4 Mathematical Formulation of Objective Functions

The primary objective in AP is to maximize the acceptability and the requirements fulfillment indices as
shown below.

Maximize the acceptability function:

$$
\alpha = \frac{\omega_i \sum_{j=1}^{n} \sum_{i=1}^{m} \mu_{ij}}{e_i} ;
$$

subject to: \( \alpha \geq \alpha_{thr} \) ; and

Maximize the requirement fulfillment function:

$$
RFF_f = \frac{\sum_{i=1}^{n} \mu_i}{n e'_i} \quad (D.5)
$$

subject to: \( RFF \geq \Phi_{thr} \)

where:

- \( \alpha \) = the acceptability index.
- \( \alpha_{thr} \) = threshold acceptability index value.
- \( RFF_f \) = fulfillment index for requirement \( f \).
- \( \Phi_{thr} \) = threshold requirement fulfillment index value.
- \( e_i \) = maximum possible score.
- \( e'_i = \Phi_{max} \sigma_{max} \rho_{max} \) = maximum possible fulfillment value.
- \( \mu \) = score.
- \( \omega_i \) = a weight assigned by the developer (requirements engineer).

For reliable AP results, it is vital to ensure that parameters that influence the accuracy of \( \alpha \) and \( RFF \)
values are optimal. In particular, relevance of the review criteria and the reliability of the members of
the review panel must be observed closely. Therefore, it is also necessary to maximize the following
objective functions as well.

Maximize the relevance function:

$$
P = \frac{\sum_{i=1}^{n} \rho_i}{n \rho_{max}} ; \quad (D.6)
$$

subject to: \( P \geq P_{thr} \) ; and

Maximize the confidence function:

$$
X = \frac{\sum_{j=1}^{n} \sum_{i=1}^{m} \chi_{ij}}{mn \chi_{max}} \quad (D.7)
$$
subject to: $X \geq X_{thr}$

where:

$P = \text{relevance index.}$

$P_{thr} = \text{threshold relevance index value.}$

$X = \text{confidence index.}$

$X_{thr} = \text{threshold confidence index value.}$

D.6 Weak Spots Analysis

Two diagrams, namely (i) suitability diagram [Figure D.5] and (ii) requirements fulfillment diagram [Figure D.6], are proposed to represent graphically the levels of fulfillment of requirements. The goal is to enable the developers quickly spot the extent to which requirements have been fulfilled and where weak spots lie. The weak spots flag the developers where the improvement efforts should be directed. By using these diagrams, the requirements that have not adequately been fulfilled can easily be recognized, and it is therefore possible to see right away differences in improvement priorities that need to be given to various requirements.

Suitability ($\sigma$) Diagram

The $\sigma$-diagrams are created based on averages and standard deviations of products of responses. The size of the error bar gives clues about how consistent the subjects have been when completing the information-gathering matrix. $\sigma$-diagrams help the developer quickly spot the extent to which requirements are fulfilled while keeping an eye on the consistency of the subjects.

Fulfillment ($\phi$) Diagram

For each requirement statement, the products of $\phi$, $\rho$, and $\chi$ are computed and compared against the maximum achievable values and the ratios plotted against the requirements. The resulting diagram is referred to as the fulfillment or $\phi$-diagram. $\phi$-diagrams differ from $\sigma$-diagrams in that they take on board raw values of the relevance of the requirements, confidences of subjects, and weights of the review criteria. They indicate which particular requirements should to be improved and thus provides means for identification of weak spots in the abstract prototypes. This is quite useful, especially if only one solution is known or when an abstract prototype falls short of scoring the $\alpha_{thr}$ value. In this case, what to improve can clearly be seen.
Appendix E: Application Case Studies

One of the key episodes in this work has been testing the usability of the AP concept. As mentioned in Chapter 6, several application case studies have been conducted in both industrial and research environments to investigate how the AP concept and software works in practice, and to determine the scope and the benefits of AP. This appendix first outlines the main AP tasks. It then describes the DSS projects involved in the case studies and details how the studies were actually conducted.

E.1 Main Activities

An AP review requires good and complete requirements as well as a good representation of the abstract prototype in question. Other issues that should also be sorted out include making a decision on the type and number of subjects to involve in AP. In some cases, in particular for the AP reviews that require hands-on experiences, an assignment must also be prepared.

Acquisition of Requirements for AP

Acquisition of requirements is a critical activity in AP. The procedure for acquisition of requirements is presented in Chapter 4. The RRP utility [refer to Chapter 5] serves as a front-end means in the acquisition of requirements. The provisional requirements are retrieved from the pool of requirements in the AP software’s database and fine-tuned (that is, irrelevant requirements discarded and requirement statements rephrased as required). Completely new requirements can be formulated and included as well. An information-gathering matrix consisting of 'recalibrated' and 'new' requirements can then be prepared. Only the requirements that relate to the applicable functional and non-functional characteristics should be included. As mentioned in Section 3.6.1, requirements can also be grouped according to their importance. For example, there are those that must be fulfilled unconditionally, others should be fulfilled as completely as possible, while certain requirements can be rather unimportant and their fulfillment can only be insisted. When formulating requirements for a DSS, it is important to envision two types of aspects, namely, technological and design aspects. Technological aspects comprise those aspects associated with computing technologies, while design aspects are those associated with product design practices. There is a big dependency between the two aspects. The envisaged system should therefore be developed while taking into the consideration these aspects.

Conceptualization and Representation of Abstract Prototypes

It is crucial for subjects, including rather novice subjects from outside the development team, to be well informed and acquainted to the abstract prototypes. To achieve this, the abstract prototypes must be represented appropriately. The steps in the process of conceptualization and representation of the abstract prototypes can be summarized as follows:

- Problem identification and clarification.
- Definition of the problem, the necessary input(s), the required output(s), general assumptions and constraints.
- Decomposition of the problem into sub-problems, maintaining contextual integrity and consistency.
- Definition of the extent of the sub-problems, pertaining input-output dependencies, specific conditions, and constraints.
- Searching for known solutions applicable to sub-problems, or creation of new solution(s).
- Harmonizing and matching alternative solutions by compatibility and connectivity analysis from a system's point of view.
- Investigation of reachability and evaluability of the individual solutions from a computational point of view.
Appendices

- Formal specification and representation of the solutions (i.e. the abstract prototypes), descriptively, symbolically, and/or diagrammatically.

Any method of representation presented in Section 4.2.2 can be adopted. To select a representation method, each of the methods in the list can, say, be evaluated relative to quick-and-dirty screening criteria established by the individual developers. The screening criteria can be, for example, the effectiveness of the method, time needed to represent the abstract prototype using the method, the availability of the supporting software tools, and memory requirements.

Development of Assignments

AP, in particular at the methods or pilot prototypes levels requires some sort of mimicking of the actual application. In this case, an assignment must be formulated or selected. The assignment must be in the right field and must focus on the problem. It should be a representative of real tasks that designers do and of design nature, and must have a certain level of complexity. It must be relatively short and manageable, but must explore the envisaged DSS thoroughly, and provide reasonable coverage of real world tasks. The assignments must be comprehensively presented to subjects. They must be well introduced and the directions as well as the tasks to be performed by subjects must be clearly stated.

Other Preparations

Before conducting the actual AP review, several other issues must be resolved. For example, the necessary number of experiments to carry out in order to have a realistic review should be determined. Furthermore, the number of subjects to involve in the review should also be decided upon, and a rating scale must be selected. All these constitute a successful AP review.

Pilot Tests

Pilot tests may be carried out before running actual AP reviews. They are conducted in order to:

- Predetermine whether the representations of the abstract prototypes and the information-gathering matrix have been well prepared and can indeed be applied successfully.
- Evaluate the assignments and to see whether they have been set correctly and whether they can be completed within the specified time.
- Check whether the directions about the assignments are sufficient for the subjects to comprehend what they have to do and to see if the AP review can proceed smoothly.
- Predetermine whether the equipment planned to be used (if any) meet the expectations and could be used as anticipated.

Running the Actual Review

In AP, the subjects often require the representations that describe what has been developed. The representations show up the dynamic and restrictive features of the eventual DSS or its parts, without the need to fully construct or install it. The following backroom preparations must be made:

- Finalization of the assignment. An assignment should be carefully finalized to take into account the improvement suggestions that may have arisen from pilot tests.
- Recruitment of the subjects. Appropriate subjects should be recruited. They should be chosen to represent typical population of stakeholders, including those who will eventually use the envisaged DSS.

Tests must focus on the designers, to understand how they will work in the framework of the proposed methods, and the influence of the new methods or software tools on their work. The designers are the arbiters of the new methods or DSS. They know what works and what do not.

In the actual AP sessions, the subjects are required to:

- Understand the proposed theories, methods, algorithms or pilot prototypes and to some extent grasp how they will ultimately solve the problem.
• Apply the implicit representation of abstract prototypes to formulate their work practice and envision how they will work when the envisaged DSS is introduced in their workplace.

• Imagine about working with the envisaged DSS and decide on whether they like it or not.

• Imagine better ways of working (if they do not like the proposed solution), transform them into implications for the design and finally express those implications clearly to the developers.

Analysis and Interpretation of Results

The general analysis scheme presented in Chapter 4 is used in the analysis of the information collected from the subjects. The principle aim is to ensure that the right solutions are adopted and the work proceeds on the right track from the beginning of the development work. The analysis scheme helps identification of weak spots and systemizes the selection of a solution or solutions from a variety of alternative solutions. During the analysis, a smaller set of solutions, which attain largest number of Φ indices, can be identified. If, for instance, two theories' Φ indices are very close, it does not necessarily always follow that the one with the highest index is the one to be preferred. Finer evaluation needs to be performed and additional review criteria need to be incorporated. Furthermore, in the process of searching for the best solutions, open discussions among the experts from the relevant fields are extremely important. The discussion topics can be, for example, on the similarities, variations, strengths and weaknesses of the tabulated solution proposals. The ultimate result of an AP review is the final design document and pilot prototypes that can be used as input in the subsequent life cycle phase, the implementation phase.

E.2 DSS Projects and Some Selected Case Studies

This section presents various experiments conducted in research and industrial environments to validate the AP concept. In both cases, reviews were carried out at theories, methods, algorithms, and pilot prototype abstraction levels.

E.2.1 Research Oriented DSS Project

The case research oriented DSS project dealt with the development of computer-based theories and methods for shape conceptualization. The specific tasks in this project included: (i) identification or formulation of various theories and methods; (ii) exploration of techniques to support shape conceptualization; and (iii) implementation and testing of software. Upon the completion of the project, a prototype Computer Aided Conceptual Design (CACD) software will be released. It is anticipated to consist of hundreds of functions for handling highly interactive communication, vague modeling and preparations for production of physical concept models.

It is expected that fully developed CACD software will assist designers to create reusable conceptual shape models. CACD software is expected to facilitate and support designers in mapping their mental images about the intended product into initial shape models. Such a system can reduce time and effort required in preparing conceptual models, and help the designers create optimum shape designs. In the most advanced state, such models can be availed in a standard format and thus provide an automated interface between conceptual and detailed design. Figure E.1 depicts the general scheme for development of various utilities of the CACD software. There are three main tasks, namely, exploratory study, reviews, and pilot implementation. The exploratory work is primarily concerned with the investigation of possible solutions using techniques such as literature surveys, interviews and brainstorming.
A function can be built based on many theories or methods, and most often several alternative theories or methods can be available for a single function. There can also be alternative algorithms or existing sets codes for a single function. The ultimate goal is to come up with consistent, complete and usable CACD software. One way of achieving this is by intensifying reviews in early phases such that faults in theories, methods, or algorithms, which forms foundations for implementation of the CACD software, can be eliminated early on, before codes are written.

Concise Overview of the Fundamental Concepts

Figure E.2 shows a process flow diagram of the proposed CACD system. There are three critical processes, namely: (i) **highly interactive communication**; (ii) **representation of vague model**; and (iii) **physical concept modeling**. The inputs of the highly interactive communication process are referred to as actions of the designer. Possible inputs to the system can be categorized as follows: (i) **materialized inputs**, for instance, scanning of physical concept models or real objects; (ii) **semi-materialized inputs**, such as shape feature insertion, imitated shape deformation, and so forth; (iii) **non-materialized inputs**, which includes inputs from means or actions such as **linguistics** (i.e. speech), **movements** (for example, gestures and spatial sketching), **haptic** and **thermal**. The interaction is defined as highly intensive since it relies on massive information and knowledge chunks accumulated in the system and allows for high-level search and matching of semantics of information. The inputs may typically be from different users that use different input techniques. Such interaction generates **knowledge chunks** related to the object, which can be processed to create an initial **natural internal representation** of the object. It is important that the detailed computer internal representation should be convertible into usable data formats. Furthermore, the representation should be monolithic in that the information must be handled using a single representation method. Converted **modeling entities** derived from the initial shape model are the inputs for physical concept modeling process in which editable physical concept models are generated to support the synthesis of shapes. It is desirable that at least portions of the physical concept model should be reusable in design iterations. Physical concept models can also be reused as inputs in other design problems to provide global shape information or local shape features. The ultimate outputs of the CACD system are computer graphic models in **natural representation**, which can be converted and used in downstream activities of embodiment and detailed design.

Extensive exploratory research was carried out to determine the desirable characteristics of the CACD system. The anticipated CACD software is intended to provide natural working environment, improve productivity by enhancing communication between the designers and shape models, and provide knowledge to designers.
Structure of the CACD System

Four modules of the CACD system have been proposed. These are: the highly interactive communication, natural internal representation, physical concept modeling and the design knowledge handling modules [Figure E.3].

- **Human Computer Interaction**

Traditionally, designers use natural techniques such as combinations of hand gestures and verbal utterance, two-dimensional shaded image sketching and claying in expressing concepts. One of the intentions of the project, has been to investigate the possibility of adopting these methods in the CACD software. While the aim has been to maintain the intuitiveness and flexibility that designers enjoy when working using traditional methods, improvement of the efficiency, productivity and applicability of the outputs of the conceptual design activities have also been the goals. A voice interface has been developed as a pioneering input means, based on the selected theories, methods and algorithms.

- **Natural Internal Representation**

The natural internal representation module provides computer-based methods for converting natural input information into formal usable representations. In order to have a complete description of a conceptual shape, there are five necessary types of information, namely: (i) **identification**, which identifies the entities that make up the object; (ii) **geometric information**, i.e. sets of points or surface patches that make up the surface of the object; (iii) **topological information**, i.e. information about the neighborhoods or connections of entities; (iv) **locational information**, which defines positions and orientations of entities; and (v) **attribute information**, i.e. materialization and visual information for example, weight and color. The ultimate aim is to develop CACD software that supports creation of initial shape representations quickly and with ample flexibility. It is desirable to ensure that shape information should not be rigidly or fully specified. For example, a surface can simply be represented using a set of sparse points or patches, location of a feature can be specified in terms of a space domain that it may occupy, and so forth. The vague discrete interval modeling (VDIM) theory [Horváth et al., 2000] has specifically been developed for the natural internal representation module.

- **Physical Concept Modeling**

The physical concept-modeling module provides techniques for preparation of manufacturing information for production of large-sized physical concept models by using the available manufacturing technologies. Large sized, free form physical concept models can be used in conceptual design of automobiles, aerodynamic devices, hydrodynamic devices and household appliances. Other areas include the entertainment industry (i.e. production of movies and stage setting in theatres) and production of advertisements. Layered prototyping techniques can be used in production of physical concept models, but the available techniques cannot be used in production of large sized models due to size limitations [Horváth, 1998]. Theories and methods that can be used in the production of large sized free-form physical concept models are being developed. The new technology is centered on the decomposition and slicing techniques [Broek et al., 1998].
Design Knowledge Handling Utility

The design knowledge handling utility provides techniques that facilitate management and processing of the design knowledge. The goal is to avail to the designers various kinds of knowledge, which represent principal ideas about the expected appearance and operation of a product. A number of design aspects such as ergonomics, aesthetics, costs, manufacturability and many others must be taken into account. Research on the development of Design for X (DFX) tools, which focuses on handling various aspects during product design, is widespread [Tichen, 1997]. However, many DFX tools in most cases handle one or few aspect only. Computer based theories and methods for handling and application of design knowledge from various different aspects for use in shape design are being developed. The focus is on formalization and structuring of knowledge related to shape conceptualization, based on the ontology and nucleus paradigms. Literature such as Jambak et al. (2002) and Pulles et al. (1999) provide more details.

E.2.1.1 Desirable Quality Characteristics

Requirements specification and allocation of features to a DSS are the activities that precede AP [see Section 3.6]. If the problem and the constraints that exist for its solution are known, the desirable characteristics can be established. The following are the main desirable characteristics for the CACD software.

- **Adaptability**: The system should automatically change to match various different real world circumstances, for instance, coping with different forms of inputs.

- **Choices of modeling techniques**: A selection of modeling techniques should be made available within the system, for example, usage of library basic shapes in building interim shapes of products, and imitation of claying.

- **Collaboration**: In many cases conceptual design tasks are performed in teams and very often customers are invited to participate. The CACD system should therefore support the designers in a single locality or in geographically remote sites to exchange ideas themselves and with customers.

- **Comprehensiveness**: The CACD system should be comprehensive. For instance, it should be able to suggest alternative solutions, ‘fill-up’ the missing data, give warnings, and so on.

- **Convenience**: The CACD system must suit the designers' needs. For instance, shape mathematics should be hidden; learning efforts should be minimal; and so forth.

- **Convertability**: The CACD system should serves as a front-end processor for the existing CAD systems. Hence, the internal representations of shapes should be convertible.

- **Creativity**: The system should allow designers to be creative.

- **Editability**: The initial product models should be editable. The CACD system should therefore be equipped with tools e.g. for positioning or cutting shape models, and so forth.

- **Flexibility**: The CACD system should allow designers to control the shape conceptualisation process. The system should not dictate the conceptualisation process, that is, it should not attempt to predetermine solutions or intermediate actions. The designers must have a chance to approve or reject any proposal put forward by the system.

- **Knowledge Intensiveness**: The CACD system should be highly knowledge intensive and should e.g. provide design suggestions, assist the designer to explore solution possibilities, etc. The response may be in the form of propositions that may typically be derived from a database or a knowledgebase containing e.g. past experience, heuristic knowledge and standard information.

- **Modality**: The CACD system should be able to treat more than one conceptual solution. It should be able to automatically create series of instances and every information item must be handled by the same representation method.

- **Reactivity**: The CACD system should be able to capture the semantics of the inputs and to react to different forms of inputs. It should be able to understand and follow what the designer thinks.
Meanwhile, the user must be free to accept or reject any suggestive reaction from the system. The designer must be involved to supplement the abilities of the system by participating in decision-making and by providing heuristics.

- **Vagueness**: The CACD system should be able to cope with e.g. domain representations, poorly defined natural surfaces, unclear or incomplete inputs and contradicting inputs.

It is desirable that shape information must be represented in a form that can be used in the embodiment and detail design. Further research work is therefore needed to solve the problems of: (i) how to represent conceptual shapes generated using natural interaction techniques; and (ii) how to convert these descriptions into standard data structure formats. It is expected that a fully developed CACD system would help the designers generate initial models intuitively and quickly. The envisaged CACD system is expected to support designers to map their mental images to initial shape models. Such a system can help reduce time and efforts required in preparing of initial models, and assist and guide the designer to generate optimum conceptual shape models. In the most advanced state, such models can be availed in a

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Review criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>The theory is strong, i.e. based on a direct law and not hypothesis</td>
</tr>
<tr>
<td>R2</td>
<td>The theory allow direct computation to be performed, having specified the necessary input</td>
</tr>
<tr>
<td>R3</td>
<td>The theory is already available in the form of software</td>
</tr>
<tr>
<td>R4</td>
<td>The theory is robust, i.e. solve a class of problems</td>
</tr>
<tr>
<td>R5</td>
<td>The theory offers effective solution</td>
</tr>
<tr>
<td>R6</td>
<td>The amount of needed input data is reasonable</td>
</tr>
<tr>
<td>R7</td>
<td>The application of the theory requires no knowledge of other related theories</td>
</tr>
<tr>
<td>R8</td>
<td>The results provided by the theory are reliable</td>
</tr>
<tr>
<td>R9</td>
<td>The theory produces precise results</td>
</tr>
<tr>
<td>R10</td>
<td>The theory can be applied in other different cases</td>
</tr>
<tr>
<td>R11</td>
<td>Numerical approximations are not needed</td>
</tr>
<tr>
<td>R12</td>
<td>The input variables are available</td>
</tr>
<tr>
<td>R13</td>
<td>The theory would be comprehensible to the programmers</td>
</tr>
<tr>
<td>R14</td>
<td>The assumptions does not adversely affect the suitability</td>
</tr>
<tr>
<td>R15</td>
<td>The eventual application would require no training on the basics of the theory</td>
</tr>
<tr>
<td>R16</td>
<td>The theory implies automatible/computerizable methods</td>
</tr>
<tr>
<td>R17</td>
<td>The theory can be used alongside other selected theories</td>
</tr>
<tr>
<td>R18</td>
<td>Application of the theory requires no human decisions/intervention</td>
</tr>
<tr>
<td>R19</td>
<td>The theory can be broken down into smaller uncoupled parts</td>
</tr>
<tr>
<td>R20</td>
<td>The theory is exhaustively described</td>
</tr>
<tr>
<td>R21</td>
<td>The theory is not based on common sense reasoning</td>
</tr>
<tr>
<td>R22</td>
<td>The implementation of the theory would require affordable time/labor investment</td>
</tr>
<tr>
<td>R23</td>
<td>The theory is in a finalized form i.e. not still under development</td>
</tr>
<tr>
<td>R24</td>
<td>The theory can be extended</td>
</tr>
<tr>
<td>R25</td>
<td>The theory can be expressed in algorithms and coded</td>
</tr>
<tr>
<td>R26</td>
<td>The conditions for application of the theory are known</td>
</tr>
<tr>
<td>R27</td>
<td>The theory does not depend on other theories</td>
</tr>
<tr>
<td>R28</td>
<td>The theory can work without failure</td>
</tr>
<tr>
<td>R29</td>
<td>The theory rightly solves the problem in hand</td>
</tr>
<tr>
<td>R30</td>
<td>Support loose shape concept</td>
</tr>
<tr>
<td>R31</td>
<td>Support conceptual shape design</td>
</tr>
<tr>
<td>R32</td>
<td>It is in harmony with thinking of stylists</td>
</tr>
<tr>
<td>R33</td>
<td>The underlying mathematics would be hidden</td>
</tr>
<tr>
<td>R34</td>
<td>It offers freedom when expressing shapes</td>
</tr>
<tr>
<td>R35</td>
<td>It is capable of handling vagueness of shapes</td>
</tr>
<tr>
<td>R36</td>
<td>It is capable of handling inconsistent input information</td>
</tr>
<tr>
<td>R37</td>
<td>It is capable of handling modality</td>
</tr>
<tr>
<td>R38</td>
<td>It is capable of handling ambiguity</td>
</tr>
<tr>
<td>R39</td>
<td>It is capable of handling under-determination</td>
</tr>
<tr>
<td>R40</td>
<td>Support handling of behavior of objects</td>
</tr>
<tr>
<td>R41</td>
<td>The user still controls the shape conceptualization process</td>
</tr>
<tr>
<td>R42</td>
<td>Maintains environment for creativity</td>
</tr>
<tr>
<td>R43</td>
<td>It is capable of handling incomplete input</td>
</tr>
<tr>
<td>R44</td>
<td>It is capable of handling contradictions</td>
</tr>
<tr>
<td>R45</td>
<td>It allows editing</td>
</tr>
<tr>
<td>R46</td>
<td>It fosters reuse of parts of the existing models</td>
</tr>
</tbody>
</table>
standard format, and thus made available for CAD systems. Recent developments in computing technologies suggest that most of the above mentioned desirable characteristics could be realised.

E.2.1.2 Some Selected Case Studies

Some selected case studies are presented in this section to give an insight into how the AP methodology was used in the research-oriented DSS development project.

E.2.1.2.1 Theories Level Abstract Prototyping

The purpose of AP at theories level is to ensure that appropriate theories are used, and that the development of DSS proceeds on the right track from the early stages of the design process. In practice, review of theories (i.e. the DSS in its highest abstraction level) is somewhat an uncommon phenomenon. This sub section presents two case studies conducted to demonstrate how AP was implemented at the theories abstraction level.

Case I: Testing Theories for the VDIM Software

The purpose of this test was to investigate the completeness and consistency of two alternative theories proposed for the VDIM software, namely the differentiable manifolds and particle system theories. To create and manipulate conceptual shapes, a theory that can lead to creation of a flexible and natural modeling technique, and that can handle fuzziness, inaccuracies and inconstancies was needed. The differentiable manifolds theory was thought to be one of the possible solutions, while the particle system theory was specially created for this purpose. The particle system theory is based on the idea of using point sets generated using highly interactive input means such as speech-based, gestures-based, and/or scanning devices. A point set model is in essence a domain-oriented representation of a cluster of shape instances. The key modeling entity is a particle, which has metric and materialistic properties, and is derived from the input devices. Several shape operators for creating objects and for manipulation of point clouds have been proposed. More details of the particle system theory are available in Rusák et al. (2000 b) and Horváth et al. (2000).

The review criteria [Table E.1] were formulated largely based on the requirements derived form the AP system's database. The review criteria were put into the standard information gathering matrix using the AP software tools. Six subjects completed and returned the information gathering forms electronically, or in paper form. All of them were members of the development team. The collected data were analyzed using the AP software, and the σ - diagrams [Figure E.4] generated. These diagrams showed the extent to which the theories fulfilled the requirements (in the opinion of the subjects), enabled the identification of weak spots in the theories, and helped the identification of the improvement priority that needed to be given to various requirements. The α indices, which compare the solutions in hand with the ideal solution were also determined, and found to be 0.49 and 0.55 for the differentiable manifolds and particle system theories respectively. The procedure for computation of the α-index has been presented in Chapter 4 and elaborated in Appendix D.

It can be seen that in many respects, the particle system theory offered a better solution. Both the α index and the σ– diagram showed that the particle system theory gives better foundation for creation of software tools for geometric modeling of vague shapes. The ṁ values were used in ranking the requirements and in deciding on an improvement strategy [see the procedure in Section 5.4.4]. It was clear from the experiment [see Figure E.4] that in order to adequately support geometric vague modeling, the particle system theory needed to be improved. In so doing, emphasis was put on ensuring fulfillment, to a satisfactory level, of the most relevant requirements (that is, those with higher ṁ values), and which at the same time attained lower values of φ indices (e.g., R5, R11, R14, R18 and R29). Improvement of the theory could lead to the creation of effective methods, algorithms and eventually implementation of high quality VDIM software.

Case II: Testing of the feature based decomposition theory

An experiment conducted to review the feature based decomposition theory, which was developed locally in a research oriented DSS project [Project Z] to deal with the problem of decomposition of CAD
models for layered slicing [Horváth et al., (2000); Broek et al., (1998)] is presented to demonstrate how the ‘no existing solution – see Section 6.3.4’ scenarios were treated. The preparations for AP included acquisition of requirements and preparation of information gathering matrix. Table E.2 shows the list of the review criteria used.

Table E.2: Review criteria used in AP of the slicing theory

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Review criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>The theory is strong, i.e. based on a direct law and not hypothesis</td>
</tr>
<tr>
<td>R2</td>
<td>The theory allows direct computation to be performed, having specified the necessary input</td>
</tr>
<tr>
<td>R3</td>
<td>The theory is already available in the form of software</td>
</tr>
<tr>
<td>R4</td>
<td>The theory is robust, i.e. solve a class of problems</td>
</tr>
<tr>
<td>R5</td>
<td>The theory offers effective solution</td>
</tr>
<tr>
<td>R6</td>
<td>The amount of needed input data is acceptable</td>
</tr>
<tr>
<td>R7</td>
<td>The application of the theory does not need knowledge of other related theories</td>
</tr>
<tr>
<td>R8</td>
<td>The results provided by the theory are reliable</td>
</tr>
<tr>
<td>R9</td>
<td>The application of the theory produce precise results</td>
</tr>
<tr>
<td>R10</td>
<td>The theory can be applied in other different cases</td>
</tr>
<tr>
<td>R11</td>
<td>Numerical approximations are not needed</td>
</tr>
<tr>
<td>R12</td>
<td>The input variables are available</td>
</tr>
<tr>
<td>R13</td>
<td>The theory can be comprehended by the programmers</td>
</tr>
<tr>
<td>R14</td>
<td>The assumptions does not adversely affect the suitability</td>
</tr>
<tr>
<td>R15</td>
<td>The eventual application would require no training on other theories</td>
</tr>
<tr>
<td>R16</td>
<td>The theory implies automatable/computable methods</td>
</tr>
<tr>
<td>R17</td>
<td>The theory can be used alongside other selected theories</td>
</tr>
<tr>
<td>R18</td>
<td>Application of the theory requires no human decisions/intervention</td>
</tr>
<tr>
<td>R19</td>
<td>The theory can be broken down into smaller uncoupled parts</td>
</tr>
<tr>
<td>R20</td>
<td>The theory is exhaustively described</td>
</tr>
<tr>
<td>R21</td>
<td>The theory is not based on common sense reasoning</td>
</tr>
<tr>
<td>R22</td>
<td>The implementation of the theory would require affordable time/labor</td>
</tr>
<tr>
<td>R23</td>
<td>The theory is in a finalized form i.e. not still under development</td>
</tr>
<tr>
<td>R24</td>
<td>The theory can be extended</td>
</tr>
<tr>
<td>R25</td>
<td>The theory can be expressed in algorithms and coded</td>
</tr>
<tr>
<td>R26</td>
<td>CAD models of various kinds of industrial products can be dealt with</td>
</tr>
<tr>
<td>R27</td>
<td>Complicated objects can be decomposed sufficiently</td>
</tr>
<tr>
<td>R28</td>
<td>The CAD model can be segmented for economic manufacturing</td>
</tr>
<tr>
<td>R29</td>
<td>The decomposed CAD model can be manufactured in a short time</td>
</tr>
<tr>
<td>R30</td>
<td>The shape of the tool can be predicted</td>
</tr>
<tr>
<td>R31</td>
<td>The number of segments can be optimized</td>
</tr>
<tr>
<td>R32</td>
<td>The tool path can be optimized</td>
</tr>
</tbody>
</table>

The theory was represented to the subjects in the form text. Nine subjects, all of them experts working in the project, participated in the experiment. Based on the collected data, $\phi$ and $\sigma$ - diagrams were created and the $\alpha$ index determined, and found to be 0.48. The later helped the developers know how closely the ideal solution had been approached. The $\phi$ and $\sigma$ - diagrams [Figure E.5] helped the developers ascertain where the weak spots lied, and where improvements and further development might be tried. It can be seen clearly from the diagrams that this theory still needed further improvements to ensure that requirements R3, R5, R7, R10, R11, R12, R14, R15, R17, R18, R19 and R31 are also adequately fulfilled\(^1\). However, fulfilment of some of the requirements that had not been adequately satisfied, for example, R3 and R17 could only be insisted.

E.2.1.2.2 Methods Level Abstract Prototyping

One of the goals of the case research-oriented DSS project was to develop a highly interactive natural interface for the CACD system. The AP technique was used to investigate the preferences of the designers on possible input techniques. The primary question was which combinations of input means

\(^1\) Judgment on whether a requirement has been fulfilled adequately can be subjective. One of the ways used to overcome this was to ensure parity in fulfillment of requirements (i.e. to ensure that all requirements were approximately fulfilled to the same level).
Figure E.4: \( \sigma \)-diagrams for the differentiable manifolds (top) and particle system (bottom) theories

are most effective. Several functional and non-functional characteristics were taken into consideration during AP. The ISO/IEC 9126 standard, which is primarily concerned with definition of characteristics used in the evaluation of software products, was used as a reference model.

Table E.3: Review criteria used in AP of the shape-input methods

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Review criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>The method allows the designer or the team of designers to work without interference</td>
</tr>
<tr>
<td>R2</td>
<td>The method match natural ways of working</td>
</tr>
<tr>
<td>R3</td>
<td>Users can understand how the input device works</td>
</tr>
<tr>
<td>R4</td>
<td>The input method is good for the intended task</td>
</tr>
<tr>
<td>R5</td>
<td>The input method can be used without the need of extensive familiarization or learning</td>
</tr>
<tr>
<td>R6</td>
<td>The input method is ergonomically OK</td>
</tr>
<tr>
<td>R7</td>
<td>The input stream consist of portable devices</td>
</tr>
<tr>
<td>R8</td>
<td>The input devices are affordable</td>
</tr>
<tr>
<td>R9</td>
<td>There is no marketing problem</td>
</tr>
<tr>
<td>R10</td>
<td>The input method can provides consistent results</td>
</tr>
<tr>
<td>R11</td>
<td>The method can solve the problem in question</td>
</tr>
<tr>
<td>R12</td>
<td>The method can be effective in actual use</td>
</tr>
<tr>
<td>R13</td>
<td>If adopted, the method will survive over a reasonable time span</td>
</tr>
<tr>
<td>R14</td>
<td>If is adopted, the method will be maintainable</td>
</tr>
<tr>
<td>R15</td>
<td>This input method can be used in the design of industrial products</td>
</tr>
<tr>
<td>R16</td>
<td>The input method will comparatively brings along more useful features over the competing methods</td>
</tr>
<tr>
<td>R17</td>
<td>The input stream can be used to communicate non-geometric shape information</td>
</tr>
<tr>
<td>R18</td>
<td>The input stream can be used to communicate geometric shape information</td>
</tr>
<tr>
<td>R19</td>
<td>The input stream can be used to input a wide range of shapes, including free-form shapes</td>
</tr>
<tr>
<td>R20</td>
<td>With the input stream, shapes can be modified</td>
</tr>
</tbody>
</table>
Table E.4: Review criteria used in AP of the points checking algorithms

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Review criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>The events are split in smaller related sub events</td>
</tr>
<tr>
<td>R2</td>
<td>It is, at any instance, clear what is to be done next</td>
</tr>
<tr>
<td>R3</td>
<td>The intermediate results of tasks are clearly defined</td>
</tr>
<tr>
<td>R4</td>
<td>The outcomes of each stage are known</td>
</tr>
<tr>
<td>R5</td>
<td>The algorithm is simple</td>
</tr>
<tr>
<td>R6</td>
<td>The algorithm is elegant</td>
</tr>
<tr>
<td>R7</td>
<td>The length of time taken to implement the algorithm is reasonable</td>
</tr>
<tr>
<td>R8</td>
<td>The algorithm terminates after a finite (required) number of steps</td>
</tr>
<tr>
<td>R9</td>
<td>Each step of the algorithm is precisely defined</td>
</tr>
<tr>
<td>R10</td>
<td>The eventual implementation would be more effective than manual methods</td>
</tr>
<tr>
<td>R11</td>
<td>There skills required for implementation are available</td>
</tr>
<tr>
<td>R12</td>
<td>The needed data can be provided</td>
</tr>
<tr>
<td>R13</td>
<td>The application of the algorithm will result in a finite set of actions</td>
</tr>
<tr>
<td>R14</td>
<td>The sequence of actions has a unique initial action</td>
</tr>
<tr>
<td>R15</td>
<td>Each action in the sequence has a successor</td>
</tr>
<tr>
<td>R16</td>
<td>The sequence terminates with either a solution to the problem or a statement that the problem is unsolvable for the set of data in question</td>
</tr>
<tr>
<td>R17</td>
<td>The algorithm is presented in a notation interpretable by the device to be used to solve the problem</td>
</tr>
<tr>
<td>R18</td>
<td>The time required to execute the algorithm is reasonably short</td>
</tr>
<tr>
<td>R19</td>
<td>The memory requirement of the algorithm is not overbearing</td>
</tr>
<tr>
<td>R20</td>
<td>The algorithm is accurate i.e. provides accurate results when used in solving the problem</td>
</tr>
<tr>
<td>R21</td>
<td>The algorithm can handle a range of input data</td>
</tr>
<tr>
<td>R22</td>
<td>The rounding or truncation errors do not accumulate to destroy the answer</td>
</tr>
<tr>
<td>R23</td>
<td>The iterations can be executed in a reasonably short time</td>
</tr>
<tr>
<td>R24</td>
<td>The algorithm is the precise characterization of a method for solving the problem</td>
</tr>
<tr>
<td>R25</td>
<td>The algorithm requests the input data (if any) and then respond as expected</td>
</tr>
<tr>
<td>R26</td>
<td>The algorithm will not halt without notification</td>
</tr>
</tbody>
</table>

This standard sets out six quality characteristics that described the CACD software quality, namely: functionality, reliability, usability, efficiency, maintainability and portability. From the AP system’s warehouse, the developers could also find several requirements, which were in turn used as review criteria [Table E.3].

By using the MDRS software, 47 requirements were found. 26 (approx. 55%) requirements were dropped right away during the fine-tuning stage, 12 requirements were taken unchanged, while only one requirement was formulated from scratch. 9 (approx. 41%) requirements in the final list of requirements were formulated based on the available requirements. What was actually done was confined to changing the wording of the available requirements to make them more specific to the task and the abstraction level. The whole exercise took less than two days.²

Having defined the requirements and metrics, the next step was to decide on what stakeholders to incorporate in AP as subject. For this particular functionality, the subjects were the designers. 21 designers, mostly industrial designers, completed and returned the information gathering matrices. The analysis was subsequently done to determine $\phi$ and $\alpha$ indices.

In the opinions of the subjects, the mouse-keyboard stream was preferred most, but nevertheless with just a slender margin. The (i) graphic pen-keyboard, and (ii) voice control-gestures-keyboard combinations followed. The differences among the $\alpha$ values for the input streams were almost insignificant. Even the trends in the $\phi$ values did not significantly differ. It was concluded that in the opinion of the selected

---

² This is relatively quite a short time. Normally formulation of requirements is quite a lengthy process that extends to, for instance, holding brainstorming sessions, recursive gathering of needs, and carrying out extensive literature search
review panel, any of the three input streams were acceptable in one way or another. Voice control and gesture, and to some extent graphic pen methods are not very common among the designers. Most subjects (designers) could not imagine precisely how to work with them, and how they would affect their working methods. This may explain why they were somehow less preferred.

This experiment gave a clue on how the input stream consisting of voice control, gestures and keyboard is relatively preferred. In the opinion of the subjects, this input stream had an edge over the others in satisfying requirements such as R2, R6, R12 and R16 [see Figure E.6]. The team could then start the next stage of the design process, namely, developing and testing algorithms and pilot prototypes.

### E.2.1.2.3 Algorithms Level Abstract Prototyping

Many algorithms for various functions of the CACD system have been created. In this section, an AP review conducted to verify the 'points checking algorithms' is presented as an example how to apply the AP methodology at the algorithms level. Two approaches (algorithms), namely 'normal inspection' and 'ray intersection' were compared to determine which one best solve the problem. By using the MDRS software, 28 requirements were found from the database. 2 of them were dropped right away during the fine-tuning exercise, 21 (80.8%) requirements were taken unchanged, while no new requirement was formulated from scratch. 5 (approx. 19.2%) requirements in the final list [Table E.4] were formulated based on the requirements derived from the database. These requirements served right away as review criteria.

The developers who also served as subjects in this exercise first investigated the validity of the review criteria (i.e. requirements) and discussed the wording and semantics of each review criterion. Some of the review criteria were dropped out for some reasons; for example, some of them seemed to be repetitive or not relevant at all. Then the subjects completed the information gathering forms
individually. Analysis was eventually conducted to determine $\phi$ and $\alpha$ indices for these algorithms. The results [Figure E.7] show that the 'normal inspection' approach was preferred most. Informal conversations with the developers after the experiment vindicated these results.

![Graph showing input streams and $\alpha$ index](image)

**Figure E.6:** Handling of several solution scenarios in AP: investigation of the designer's input device preferences

**E.2.1.2.4 Pilot Prototypes Level Abstract Prototyping**

Several pilot prototypes have been implemented. The pilot implementations of the VDIM software, speech recognition software, and of a 3D points input UI are used as examples to illustrate how the AP concept was used at the pilot prototypes abstraction levels. Preparatory work included acquisition of requirements and preparation of the information gathering matrices. By using the AP software, 42 requirements were found from the warehouse. The list was reviewed and 27 (64%) of them were dropped for not being relevant. No new requirement was formulated from scratch or based on the requirements derived from the warehouse. The developers used these requirements as review criteria, which were subsequently used as a checklist for investigating the completeness of pilot prototypes. Table E.5 shows the list of the review criteria used.
Table E.5: Review criteria used in AP of pilot prototypes

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Review criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>All critical features have been incorporated</td>
</tr>
<tr>
<td>R2</td>
<td>The implementation work without failure</td>
</tr>
<tr>
<td>R3</td>
<td>The implementation is effective</td>
</tr>
<tr>
<td>R4</td>
<td>The implementation is engineered according to all pre set specifications</td>
</tr>
<tr>
<td>R5</td>
<td>The implementation is portable across platforms</td>
</tr>
<tr>
<td>R6</td>
<td>All repetitive tasks are automated</td>
</tr>
<tr>
<td>R7</td>
<td>The implementation can use data from other software that it depends upon</td>
</tr>
<tr>
<td>R8</td>
<td>The application operations are not tied to the underlying theories or methods</td>
</tr>
<tr>
<td>R9</td>
<td>The output data format is compatible to those of downstream applications</td>
</tr>
<tr>
<td>R10</td>
<td>The implementation adequately support the targeted tasks</td>
</tr>
<tr>
<td>R11</td>
<td>The implementation can be kept in use for the specified period of time</td>
</tr>
<tr>
<td>R12</td>
<td>The associate technologies are sustainable</td>
</tr>
<tr>
<td>R13</td>
<td>The proposed solution address the project goals</td>
</tr>
<tr>
<td>R14</td>
<td>The output data is as precise as required</td>
</tr>
<tr>
<td>R15</td>
<td>The response time is according to the specifications</td>
</tr>
</tbody>
</table>

Figure E.8 summarizes the opinion of the respective developers on the implementations. Clearly, as can be seen, all three pilot prototypes had to be improved further to sufficiently address certain requirements (such as R3, R11, R12 and R14 for the 3D points input UI; R2, R4, R5, R11, and R12 for the speech recognition software; and R2, R5 and R6 for the VDIM software) that were not fulfilled to the levels of other requirements.

![Graph](image)

(a) $\phi$ - diagram

![Graph](image)

(b) Acceptability of the algorithms

Figure E.7: $\phi$ diagram for the normal inspection and the ray intersection algorithms
E.2.2 Industrial DSS Project

AP was deployed in the industrial DSS project and provided: (i) a model for directing and shaping design activities, (ii) a technique for crosschecking fulfillment of requirements at the design phase, and (iii) a methodology for identification of potential glitches in the envisaged Photogrammetry software [see also Section 6.3.1]. As for the two later application orientations, the AP involved the following activities.

1. Acquisition of requirements for the Photogrammetry software.
2. Classification of requirements according to the abstraction levels. Some requirements could be tested even as early as at theories abstraction level, or at methods or algorithms level, while others required executable implementations.
3. Identification of features for the Photogrammetry software.
4. Identification or creation of initial solutions (that is, suitable theories, methods, algorithms or pilot prototypes).
5. Formal representation of solutions.
6. Reviewing initial solutions by using the AP methodology. This included determination of \( \alpha \) and \( \phi \) indices and creation of \( \phi \) or \( \sigma \) - diagrams.
7. Repeating steps 5 and 6 in the subsequent levels.

As can be seen, the initial activities (that is, step 1 through to 4) should not necessarily be accomplished in a strict, singular order.

E.2.2.1 Acquisition and Sorting of Requirements According to the Abstraction Levels

Acquisition of requirements was a two front activity. Some of the requirements were formulated based on needs analysis, while others were obtained from the AP software's database. The later kinds of requirements were mainly those used in previous DSS project (that is, in the research-oriented DSS project – see Section E.2.1). The requirements from the database were in a semi-processed form, in that they were already sorted according to the abstraction levels, while those that originated from needs analysis had to be sorted out. A total of 84 requirements (23, 20, 26, and 15 requirements testable at theories, methods, algorithms, and pilot prototypes abstraction levels respectively) were obtained from the database, while 25 requirements were formulated afresh, based on needs analysis. The needs analysis based requirements were also eventually grouped according to the abstraction levels in which they can be tested. 7 of them could be tested as early as from theories abstraction level, while 13 of them as early as from methods level. All of the needs analysis based requirements could be tested both at the algorithms and pilot prototypes abstraction levels. Sorted out requirements were then appraised and finally the lists reduced to 24, 29, 43, and 39 requirements testable at theories, methods, algorithms, and pilot prototypes.
abstraction levels respectively. These were later converted into review criteria as shown in Tables E.6, E.7, E.8, and E.9.

**Table E.6: Review criteria used in AP of the industrial case study theories**

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Review criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>The theory is strong, i.e. based on a direct law (i.e. mature and proved law) and not hypothesis</td>
</tr>
<tr>
<td>R2</td>
<td>The theory is robust, i.e. solve a class of problems</td>
</tr>
<tr>
<td>R3</td>
<td>The use of the theory allows effective calculations</td>
</tr>
<tr>
<td>R4</td>
<td>The application resulting from the theory will not require the user to have knowledge of the theory</td>
</tr>
<tr>
<td>R5</td>
<td>The results provided by the theory are reliable</td>
</tr>
<tr>
<td>R6</td>
<td>The application of the theory can produce precise results</td>
</tr>
<tr>
<td>R7</td>
<td>The theory is universal</td>
</tr>
<tr>
<td>R8</td>
<td>The number of input parameters is reasonable</td>
</tr>
<tr>
<td>R9</td>
<td>The theory is comprehensible to the programmers</td>
</tr>
<tr>
<td>R10</td>
<td>The associated assumptions do not adversely affect the suitability (i.e. accuracy, etc)</td>
</tr>
<tr>
<td>R11</td>
<td>The theory acceptable to the stakeholders</td>
</tr>
<tr>
<td>R12</td>
<td>The theory can be converted into methods</td>
</tr>
<tr>
<td>R13</td>
<td>The theory can be used in other similar tasks</td>
</tr>
<tr>
<td>R14</td>
<td>The application based on the theory would not involve excessive human decisions/interventions</td>
</tr>
<tr>
<td>R15</td>
<td>The theory is exhaustively described</td>
</tr>
<tr>
<td>R16</td>
<td>The implementation of the theory require reasonable time/labour (cost of realization is affordable)</td>
</tr>
<tr>
<td>R17</td>
<td>The theory is in a finalized form i.e. not still under development</td>
</tr>
<tr>
<td>R18</td>
<td>Suitable for small and large vessels</td>
</tr>
<tr>
<td>R19</td>
<td>Possibility to process vessels that don't have an unobstructed view of the entire hull</td>
</tr>
<tr>
<td>R20</td>
<td>The number of control points needed is as small as possible and involvement of the user is relatively less extensive</td>
</tr>
<tr>
<td>R21</td>
<td>Possibility to work without a fixed set-up of equipment while taking photo's</td>
</tr>
<tr>
<td>R22</td>
<td>High accuracy</td>
</tr>
<tr>
<td>R23</td>
<td>Structured output, i.e. no ‘point cloud’</td>
</tr>
<tr>
<td>R24</td>
<td>Suitable for all types of vessels</td>
</tr>
</tbody>
</table>

**E.2.2.2 Software Features**

The industrial DSS project involved development of various software tools, some of which were of peripheral importance as far as focus of the case study was concerned. The interest was on development of a module for re-engineering of the shape of ship hull.

![Diagram](image_url)

**Figure E.9:** (a) $\phi$ - diagram and (b) $\alpha$ indices for candidate theories
The inputs into the prospective software were photographs of specially marked spots taken from ship hull, which had to in turn be processed and used in regeneration of real world coordinates. The expected output of the module was a description of a hull shape in the form of a structured list of coordinates. The design process started by finding the existing abstract prototypes that could be used as a starting point in designing the core functions of the Photogrammetry software. Algorithms and codes for the majority of functions were readily available. However, for some functions, no existing codes, algorithms or methods that could directly be used were available. In this case, it was necessary to start from scratch, at the theories abstraction level, and follow the TMAP path.

E.2.2.3 Review Processes

The design process passed sequentially through all four stages implicating four different abstraction levels of the Photogrammetry software. The activities involved included: (a) identification, review and selection of theories; (b) formulation, review and selection of methods; (c) creation of algorithms; and (d) writing codes for a pilot prototype.

E.2.2.3.1 Prototyping and Review of Theories

There are several kinds of theories in the literature and anyone could be used as a foundation for the Photogrammetry software. Clearly not all of them are of equal quality, as far as the problem is concerned. Based on the knowledge and the expert judgments of the developers, the following theories were chosen, and given serious consideration. These are:

- photogrammetry theory,
- laser scanning theory, and
- physical measuring theory.

All of these are mature theories, and are well documented in the literature\(^3\). Each of these theories had some shortcomings as far as the problem is concerned, the main ones being as follows.

- The photogrammetry theory required control points.
- The laser scanning theory was deemed to be expensive to realize since it requires expensive equipment.

\(^3\) In the context of AP, textual description of a theory is a representation of the envisaged software, and it provides the feel and look of what the DSS will be like.
The physical measuring theory requires intensive involvement of the user and is therefore not suitable for large vessels.

These three theories were reviewed in the framework of the AP concept. The review criteria in Table E.6 were used. The developers played the role of subjects. There was no need of formal representations because they were sufficiently knowledgeable about the theories. The information gathered was eventually analyzed, and for each alternative α index determined, and ϕ - diagram plotted.

The results show that the Photogrammetry theory had an edge over other theories in fulfilling key requirements, including R1, R12, R17, R18, R19, R23 and R24. Also, its α index was comparatively higher. Thus, the Photogrammetry theory eventually emerged as the most suitable theory as it met most of the needs, and based on it various methods for parametric re-engineering of ship hull were developed.

E.2.2.3.2 Prototyping and Review of Methods

Theories imply methods, which in turn forms foundations for designing algorithms. In the context of AP, the better the methods, the higher the chance of implementing high quality software. Methods for putting the Photogrammetry theory into practice were thought of, and three alternative methods considered to be feasible were developed. These are:

- bundle block adjustment in combination with direct linear transformation [Figure E.10];
- bundle block adjustment in combination with spatial resection and intersection [Figure E.11] and
- bundle block adjustment in combination with analog field measurements [Figure E.12].

These methods had the following disadvantages as far as the problem is concerned:

- For the 'bundle block adjustment in combination with spatial resection and intersection' method, at least two photographs with all markers were required.
- The 'bundle block adjustment in combination with direct linear transformation' required a large number of control points.
- The 'bundle block adjustment in combination with analog field measurements' required field measurements to be taken.

**Figure E.11:** Bundle block adjustment with spatial resection and intersection

**Figure E.12:** Bundle block adjustment with analog field measurements
Representations of these methods were prepared\(^4\) and all three methods reviewed. The review criteria shown in Table E.7 were used. These methods are highly technical, and only the developers could sufficiently evaluate them.

![Diagram](image)

(a)

![Diagram](image)

(b)

Figure E.13: (a) \(\phi\) - diagram and (b) acceptability indices for candidate methods

Therefore, the developers played the role of subjects, and completed the information gathering matrices. The collected data was analyzed and for each alternative, \(\alpha\) index determined, and \(\phi\) - diagram generated. As can be seen from the \(\phi\) - diagram [see Figure E.13], the 'bundle block adjustment in combination with spatial resection and intersection' method fulfilled most of requirements better than other methods. Its \(\alpha\) index was also evidently higher. After thorough expert judgment, this method was formally selected and used as a foundation for creating algorithms for the Photogrammetry software.

### E.2.2.4 Implementation and Testing of the Selected Method

The bundle block adjustment in combination with spatial resection and intersection method [Figure E.11] provided the foundation for creating algorithms for the Photogrammetry software. Most codes for the envisaged software were readily available, except for the 'first spatial resection', 'spatial intersection', and 'the second spatial intersection' functions. The main task during coding was therefore to implement these functions. Until time of publication of this thesis, the implementation was still going on. It was planned that after coding, the review criteria in Table E.8 would be used as criteria for testing the pilot implementation. It was planned that various stakeholders, including the end-users, would be asked to

---

\(^4\) Representation of methods diagrammatically is one way of availing methods for AP. Other ways of representation are presented in Section 4.2.2.
experiment with the prototype Photogrammetry software, and to fill the information gathering matrices. The collected data would eventually be analyzed, $\alpha$ index determined, and $\phi$ – diagram generated.

Table E.7: Review criteria used in AP of industrial case study methods

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Review criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>The method can be computerized (i.e., converted into software).</td>
</tr>
<tr>
<td>R2</td>
<td>The method can be used without the need of extensive familiarization or learning.</td>
</tr>
<tr>
<td>R3</td>
<td>The eventual software can survive in a reasonable time span.</td>
</tr>
<tr>
<td>R4</td>
<td>The required implementation technologies are available.</td>
</tr>
<tr>
<td>R5</td>
<td>The facilities and skills for implementation available.</td>
</tr>
<tr>
<td>R6</td>
<td>The method can solve the intended problem.</td>
</tr>
<tr>
<td>R7</td>
<td>The sub-methods are compatible.</td>
</tr>
<tr>
<td>R8</td>
<td>The activities/sub-activities required to reach at the final goals are well defined.</td>
</tr>
<tr>
<td>R9</td>
<td>The inputs of each activity/sub-activity are known.</td>
</tr>
<tr>
<td>R10</td>
<td>The outputs of each activity/sub-activity are known.</td>
</tr>
<tr>
<td>R11</td>
<td>The implied algorithm can be implemented.</td>
</tr>
<tr>
<td>R12</td>
<td>The results would be precise enough.</td>
</tr>
<tr>
<td>R13</td>
<td>The order of events is sensible.</td>
</tr>
<tr>
<td>R14</td>
<td>The stages of the supported process are clearly defined.</td>
</tr>
<tr>
<td>R15</td>
<td>The solution is consistent with the goals.</td>
</tr>
<tr>
<td>R16</td>
<td>The method can be implemented into software within the requested time.</td>
</tr>
<tr>
<td>R17</td>
<td>Taking into account calibration data of a specific camera.</td>
</tr>
<tr>
<td>R18</td>
<td>Capable of giving satisfying results with metric and non-metric cameras.</td>
</tr>
<tr>
<td>R19</td>
<td>Capable of coming to a realistic solution without or with minimum user interference in iterative processes.</td>
</tr>
<tr>
<td>R20</td>
<td>Scaling and positioning of the model by using absolute coordinates or relative distances between markers or control points.</td>
</tr>
<tr>
<td>R21</td>
<td>Free choice in positioning and directing of the global orthogonal coordinate system.</td>
</tr>
<tr>
<td>R22</td>
<td>The number of control points needed is as small as possible (as small amount of photographs as possible).</td>
</tr>
<tr>
<td>R23</td>
<td>Suitable for small and large vessels.</td>
</tr>
<tr>
<td>R24</td>
<td>Possibility to process vessels that don’t have an unobstructed view on the entire hull.</td>
</tr>
<tr>
<td>R25</td>
<td>Possibility to work without a fixed set-up of equipment while taking photo’s.</td>
</tr>
<tr>
<td>R26</td>
<td>High accuracy.</td>
</tr>
<tr>
<td>R27</td>
<td>Structured output, i.e., no ‘point cloud’.</td>
</tr>
<tr>
<td>R28</td>
<td>Suitable for all types of vessels.</td>
</tr>
<tr>
<td>R29</td>
<td>Possibility to process large amounts of data (photos, control points, markers).</td>
</tr>
</tbody>
</table>

E.2.3 Applications of Specific Tools of the AP Software

The RRP and the KRP utilities played vital roles of availing two critical ingredients of AP, namely the (i) review criteria, and (ii) representations of the abstract prototypes. This section discusses how these utilities have been used in AP.

E.2.3.1 RRP Utility

There is a clear indication from the case studies that the RRP utility can be a very helpful software tool in processing requirements. The statistics on the use of this utility [Figure 6.8] clearly show that large percentages of the requirements taken from the database are used unchanged, and significant percentages are also used subject to only minor modifications, namely after only changing the wordings to make them more understandable or specific to the problem in question, without changing the semantics of the requirement statements. Very few new requirements were formulated from scratch. Having well-formulated and relevant requirements in the database is central to the success of this computer based requirements acquisition concept. The AP methodology coupled good requirements and a suitable development model can lead to development of a DSS with the desired level of quality. It can be said that the RRP tool reduced the risk of overlooking requirements. It enabled the developers to quickly collect and take into consideration a wide spectrum of requirements.

Experiences during the application case studies show that joint review of the review criteria is vital. This helped subjects to precisely understand the review criteria. With right review criteria, well understood by the developers as well as the subjects, there was a greater chance of having reliable AP results. The RRP utility had also been useful for other purposes. For example, it provided means for cross checking various kinds of information about the requirements. The developer could, for instance, trace the origin
of a requirement, the stakeholder that requires it, whether or not a requirement is specific or generic, if a requirement is applicable in other levels or phases as well, and so on.

Table E.8: Review criteria used in reviewing algorithms

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Review criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>It is at any instance clear what is to be done next</td>
</tr>
<tr>
<td>R2</td>
<td>The intermediate results of tasks are clearly defined</td>
</tr>
<tr>
<td>R3</td>
<td>The outcomes of each stage are known</td>
</tr>
<tr>
<td>R4</td>
<td>The algorithm is simple</td>
</tr>
<tr>
<td>R5</td>
<td>The algorithm is elegant</td>
</tr>
<tr>
<td>R6</td>
<td>The algorithm terminates after a finite (required) number of steps</td>
</tr>
<tr>
<td>R7</td>
<td>Each step of the algorithm is precisely defined</td>
</tr>
<tr>
<td>R8</td>
<td>There is adequate implementation skills</td>
</tr>
<tr>
<td>R9</td>
<td>The needed input data can be provided</td>
</tr>
<tr>
<td>R10</td>
<td>The application of the algorithm will results in a finite set of actions</td>
</tr>
<tr>
<td>R11</td>
<td>The sequence of actions has a unique initial action</td>
</tr>
<tr>
<td>R12</td>
<td>Each action in the sequence has a successor</td>
</tr>
<tr>
<td>R13</td>
<td>The sequence terminates with either a solution to the problem, or a statement that the problem is unsolvable for the set of data in question.</td>
</tr>
<tr>
<td>R14</td>
<td>The algorithm is presented in a notation interpretable by the device to be used to solve the problem</td>
</tr>
<tr>
<td>R15</td>
<td>The memory requirement of the algorithm is not overbearing</td>
</tr>
<tr>
<td>R16</td>
<td>The algorithm is accurate i.e. provides accurate results when used in solving the problem</td>
</tr>
<tr>
<td>R17</td>
<td>The algorithm can handle a range of input data</td>
</tr>
<tr>
<td>R18</td>
<td>The rounding or truncation errors do not accumulate to destroy the answer</td>
</tr>
<tr>
<td>R19</td>
<td>The iterations can be executed in a reasonably short time</td>
</tr>
<tr>
<td>R20</td>
<td>The algorithm is the precise characterization of a method for solving the problem</td>
</tr>
<tr>
<td>R21</td>
<td>The algorithm request the input data (if any) and then respond as expected</td>
</tr>
<tr>
<td>R22</td>
<td>The algorithm will not halt without notification</td>
</tr>
<tr>
<td>R23</td>
<td>Integration with Pies and Fairway³</td>
</tr>
<tr>
<td>R24</td>
<td>Automatic detection of faulty in- or output</td>
</tr>
<tr>
<td>R25</td>
<td>Clear separation between in- and output in the user interface</td>
</tr>
<tr>
<td>R26</td>
<td>Output data has to be imported into Fairway (SXF format)</td>
</tr>
<tr>
<td>R27</td>
<td>Suitable for small and large vessels</td>
</tr>
<tr>
<td>R28</td>
<td>Possibility to process vessels that don’t have an unobstructed view on the entire hull</td>
</tr>
<tr>
<td>R29</td>
<td>Possibility to work without a fixed set-up of equipment while taking photo’s</td>
</tr>
<tr>
<td>R30</td>
<td>High accuracy</td>
</tr>
<tr>
<td>R31</td>
<td>Structured output, i.e. no “point cloud”</td>
</tr>
<tr>
<td>R32</td>
<td>Possibility to process large amounts of data (photos, control points, markers)</td>
</tr>
<tr>
<td>R33</td>
<td>Suitable for all types of vessels</td>
</tr>
<tr>
<td>R34</td>
<td>Capable of coming to a realistic solution without or with minimum user interference in iterative processes</td>
</tr>
<tr>
<td>R35</td>
<td>Minimum use of algorithms that are inclined to run into numerical problems</td>
</tr>
<tr>
<td>R36</td>
<td>Marker positioning on sub-pixel level</td>
</tr>
<tr>
<td>R37</td>
<td>Semi automatic marker matching</td>
</tr>
<tr>
<td>R38</td>
<td>Capable of giving satisfying results with metric and non-metric cameras</td>
</tr>
<tr>
<td>R39</td>
<td>Taking into account calibration data of a specific camera</td>
</tr>
<tr>
<td>R40</td>
<td>Scaling and positioning of the model by using absolute coordinates or relative distances between markers or control points</td>
</tr>
<tr>
<td>R41</td>
<td>Free choice in positioning and directing of the global orthogonal coordinate system</td>
</tr>
<tr>
<td>R42</td>
<td>Automatic computing of initial approximations for camera positions and rotations</td>
</tr>
<tr>
<td>R43</td>
<td>The number of control points needed is as small as possible</td>
</tr>
</tbody>
</table>

Case studies only gave insights into how to use the RRP utility in AP. It should be noted that a comprehensive AP review is a much more complex process. For instance, the requirements used in case studies have been formulated by considering only the known desirable characteristics and might probably ignored e.g. the requirements for the larger systems of which the DSS systems were to be part of.

E.2.3.2 KRP Utility

As mentioned in Chapter 5, the KRP utility has been developed to provide the subjects and the developers of DSS with means for handling knowledge about abstract prototypes. The abstract prototypes were progressively developed, and their representations created and stored in the database. Over two hundred representations (mainly of foundational theories, underlying methods, algorithms and

³ Refer to Koelman, 1999.
Table E.9: Review criteria used in AP of pilot prototype during the industrial case study

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Review criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>All critical features are available</td>
</tr>
<tr>
<td>R2</td>
<td>The implementation work without failure.</td>
</tr>
<tr>
<td>R3</td>
<td>The implementation is advantageous functional wise compared to manual practices.</td>
</tr>
<tr>
<td>R4</td>
<td>The implementation is engineered according to all pre set specifications</td>
</tr>
<tr>
<td>R5</td>
<td>All repetitive tasks are automated</td>
</tr>
<tr>
<td>R6</td>
<td>The implementation can use data from other software that it depends upon</td>
</tr>
<tr>
<td>R7</td>
<td>The application operations are not tied to the underlying theones or methods.</td>
</tr>
<tr>
<td>R8</td>
<td>The output data format is compatible to those of downstream applications</td>
</tr>
<tr>
<td>R9</td>
<td>The implementation adequately support the specified tasks</td>
</tr>
<tr>
<td>R10</td>
<td>The implementation can be kept in use for the specified period of time</td>
</tr>
<tr>
<td>R11</td>
<td>The underlying implementation technologies are sustainable</td>
</tr>
<tr>
<td>R12</td>
<td>The goals and the proposed solutions are in harmony</td>
</tr>
<tr>
<td>R13</td>
<td>The input data can be provided</td>
</tr>
<tr>
<td>R14</td>
<td>The response time is according to the specifications</td>
</tr>
<tr>
<td>R15</td>
<td>Efficient running of the module</td>
</tr>
<tr>
<td>R16</td>
<td>Stable running of the module</td>
</tr>
<tr>
<td>R17</td>
<td>Integration with Pias and Fairway</td>
</tr>
<tr>
<td>R18</td>
<td>Clear separation between in- and output in the user interface</td>
</tr>
<tr>
<td>R19</td>
<td>Easy and fast data importing and processing by the user</td>
</tr>
<tr>
<td>R20</td>
<td>Automatic detection of faulty in- or output</td>
</tr>
<tr>
<td>R21</td>
<td>User friendly</td>
</tr>
<tr>
<td>R22</td>
<td>Output data has to be imported into Fairway (SXF format)</td>
</tr>
<tr>
<td>R23</td>
<td>Suitable for small and large vessels</td>
</tr>
<tr>
<td>R24</td>
<td>Possibility to process vessels that don’t have an unobstructed view on the entire hull</td>
</tr>
<tr>
<td>R25</td>
<td>Possibility to work without a fixed set-up of equipment while taking photo’s</td>
</tr>
<tr>
<td>R26</td>
<td>High accuracy</td>
</tr>
<tr>
<td>R27</td>
<td>Structured output, i.e. no ‘point cloud’</td>
</tr>
<tr>
<td>R28</td>
<td>Possibility to process large amounts of data (photos, control points, markers)</td>
</tr>
<tr>
<td>R29</td>
<td>Suitable for all types of vessels</td>
</tr>
<tr>
<td>R30</td>
<td>Capable of coming to a realistic solution without or with minimum user interference in iterative processes</td>
</tr>
<tr>
<td>R31</td>
<td>Minimum use of algorithms that are inclined to run into numerical problems</td>
</tr>
<tr>
<td>R32</td>
<td>Marker positioning on sub-pixel level</td>
</tr>
<tr>
<td>R33</td>
<td>Semi automatic marker matching</td>
</tr>
<tr>
<td>R34</td>
<td>Capable of giving satisfying results with metric and non-metric cameras</td>
</tr>
<tr>
<td>R35</td>
<td>Taking into account calibration data of a specific camera</td>
</tr>
<tr>
<td>R36</td>
<td>Scaling and positioning of the model by using absolute coordinates or relative distances between markers or control points</td>
</tr>
<tr>
<td>R37</td>
<td>Free choice in positioning and directing of the global orthogonal coordinate system</td>
</tr>
<tr>
<td>R38</td>
<td>Automatic computing of initial approximations for camera positions and rotations</td>
</tr>
<tr>
<td>R39</td>
<td>The number of control points needed is as small as possible</td>
</tr>
</tbody>
</table>

Few pilot prototypes were to the date of publication of this thesis available in the database. Apart from the storage role, the KRP utility offered means for processing information. The abstract prototypes database was made available across the networks. This enabled the subjects from different areas of expertise, regardless of their locations, to quickly and effectively access information. The subjects in remote locations could therefore read or listen to the facts, see graphical representations, or run the executable implementations. This enabled them to have better understanding of the abstract prototypes and increased the chance of achieving accurate assessments. The MDKS utility helped the developers in tracing the relationships among abstract prototypes in different abstraction levels and in exploring the implications of choices. This was quite useful, especially during synthesis, when, for instance, the consequences of adopting an alternative were being explored.

The KRP utility also provided effective means for presentation, browsing and visualization of abstract prototypes. It can be said that the KRP utility enabled the subjects to become more knowledgeable and able to make accurate judgments during AP.
Summary

This thesis describes the research work on the development of a theory and a computer based methodology for pre-implementation testing of software used in engineering design of artificial products, collectively referred to as design support software (DSS). The general research issue is that, while the significance of the DSS in day-to-day lives of humans is pretty evident, and while software used in other professions or in providing services have long been a concern in the software engineering community, and there has been a great deal of research work on quality assurance of these software, paradoxically, the DSS have not received the same attention. Research in software engineering fields have resulted into creation of numerous quality assurance measures, but nevertheless, the software engineering community in general concedes that none of them is a silver bullet strategy that works for all software organizations and for all software projects. In general terms, quality assurance is still a big challenge to software developers, and publications show that in a typical development process, the design phase accounts for significant number of faults in software products. This research addressed this problem and the focus had been on quality assurance of the DSS in the design phase of their development process.

The hypothesis in this work has been that the process of designing DSS passes through various stages, implicating distinct abstraction levels and contexts, and yields various different in-process implementations. Furthermore, it had been hypothesized that faults in these in-process implementations can be spotted in an orderly manner and various stakeholders can systematically be involved in this. The method used to prove the hypothesis can be describes as follows. A literature survey and empirical investigations were first carried out to study the strategies used in quality assurance of DSS, to understand what actually happens during the development of DSS, and subsequently to conclude, based on the results, on a theory and a methodology for pre-implementation testing of DSS. The theories and methods were elaborated, and afterwards software tools, which support the pre-implementation testing activities, were developed. The novel concept and software were subsequently applied in real world situations to study their validity. This work has resulted into the following:

- Creation of the abstract prototyping (AP) concept. In the framework of this concept, reviews are interwoven into the DSS design process. This not only locates faults during the design of DSS, but also gives some level of confidence that the development process proceeds on the right track. The concept assumes an iterative incremental approach in which the envisaged DSS is repeatedly prototyped and reviewed in different abstraction levels. Theories, methods, algorithms and pilot prototypes are recognized as testable in-process implementations. These in-process implementations have been named abstract prototypes. Requirements play the central role in AP in that they are clustered according to the abstraction levels and serve indirectly as review criteria. The representatives of various stakeholders are systematically involved in the review of the abstract prototypes.

- Development of prototype software tools, which support the developers of the DSS in AP. At the initial stages of the AP process, software supports preparation activities, namely (i) acquisition of requirements, (ii) authoring of the representations of the abstract prototypes, (iii) providing expert guidance, for instance, during the selection of the forms of representations of the abstract prototypes, and in the selection of subjects, and (iv) availing the AP knowledge and specific execution guidelines. At the final stages of the AP process, software supports processing of field information. The AP software also allows for the requirements, the representations of the abstract prototypes, and the analysis results to be stored for use during the DSS development interval or for reuse in future projects. Also, software facilitates the AP process by providing specific templates and links to various information sources.

- Practical application of the AP methodology and software in the processes of development and reviewing the abstract prototypes. Most of the developers and subjects who applied the AP methodol-
ology deemed it as a natural and logical way of designing and ensuring quality and trustworthiness of DSS. Case studies demonstrated that AP (i) offers a sound framework for shaping and directing the DSS design activities, (ii) provides a systematic method for involving various stakeholders in checking progressively if requirements are being met satisfactorily, and (iii) equips the developers with means for measuring how well the abstract prototypes are being developed. Metrics and measurements allow for systematic selection of solutions, help identification of weak spots in the abstract prototypes, and offer means for early recognition of the needs for enhancements. In relation to requirements engineering, AP (a) facilitated acquisition of requirements; and (b) provided means for measuring how well the requirements have been fulfilled as well as for establishing priorities that need to be given to various requirements.

The main achievements of this research work can be summarized as revelation and vindication that the developers work with the DSS in the design phase in four distinct guises, namely theories, methods, algorithms, and pilot prototypes, and creation of:

- a theory and a computer based methodology for pre-implementation testing of the DSS;
- a model for directing and shaping activities in the DSS design phase;
- a systematic procedure for involving various stakeholders in the design of the DSS; and
- a framework for classifying and testing the fulfillment of requirements when designing DSS, predominantly without the necessity of writing codes.

It can be said that AP reduces susceptibility of the abstract prototypes to errors, and that bad decisions, which can otherwise jeopardize the DSS project or adversely affect the quality and trustworthiness of the eventual DSS cannot be institutionalized. It promotes consideration of aims, possibilities, and various quality characteristics in a systematic way early on, and helps to reduce imperfections that often lead to rework once a DSS is judged to be poorly engineered. As a result of application of this computer-based methodology, proven and reliable theories, methods and algorithms can be used in the creation of DSS. The application of this methodology can ultimately warrant realization of usable and low risk DSS.
Samenvatting

 Dit proefschrift beschrijft het onderzoekswerk naar de ontwikkeling van een theorie en een computergebaseerde methodologie voor het testen van software die wordt gebruikt bij het ontwerpen van artefacten, nog voordat deze software geïmplementeerd is. Dergelijke software wordt in dit proefschrift Design Support Software (DSS) genoemd. Het algemene onderzoeksprobleem is de paradox dat, terwijl de betekenis van DSS in het dagelijkse leven van mensen vrij duidelijk is, de software die wordt gebruikt bij andere professies of bij het leveren van diensten, lange tijd de aandacht heeft gekregen van de softwareontwikkelingsgemeenschap en er veel werk is verricht ten behoeve van de kwaliteitsborging van deze software en dat DSS niet dezelfde aandacht heeft gekregen. Onderzoek op het gebied van software engineering heeft geresulteerd in een groot aantal kwaliteitsborgingsmaatstaven, maar desalniettemin geeft de softwareontwikkelingsgemeenschap in het algemeen toe dat geen van deze maatstaven een wondermiddel is dat werkt voor alle softwareorganisaties en voor alle softwareprojecten. In algemene termen is kwaliteitsborging nog steeds een grote uitdaging voor softwareontwikkelaars en publicaties tonen aan dat in een kenmerkend ontwikkelingsproces, de ontwerpfase verantwoordelijk is voor een significant aantal fouten in softwareproducten. Dit onderzoek richt zich op dit probleem en heeft zich geconcентreerd op kwaliteitsborging van DSS in de ontwerpfase van het ontwikkelingsproces.

 De hypothese in dit werk is dat het proces van DSS-ontwerpen verschillende stadia kent, met daarin verschillende abstractieniveaus en contexten en resulteert in verschillende in-process implementaties. Verder is gesteld dat fouten in deze in-process implementaties op een ordelijke wijze kunnen worden gesignaleerd en dat verschillende belanghebbende hierbij systematisch kunnen worden betrokken. De methode die is gebruikt om de hypothese te bewijzen kan als volgt worden beschreven. Eerst zijn een literatuurstudie en een empirisch onderzoek uitgevoerd om de stand van zaken met betrekking tot de strategieën die bij de kwaliteitsborging van DSS worden gebruikt, te bestuderen om te begrijpen wat er feitelijk gebeurt tijdens de ontwikkeling van DSS. Vervolgens zijn op basis van de resultaten conclusies getrokken ten aanzien van een theorie en een methodologie voor het testen van DSS waarbij deze software is geïmplementeerd. De theorieën en methoden zijn uitgewerkt en daarna zijn softwaregereedschappen ontwikkeld die deze testactiviteiten ondersteunen. Het nieuwe concept en de nieuwe software zijn vervolgens toegepast in reële situaties om hun bruikbaarheid te testen. Dit werk heeft geresulteerd in het volgende:

- Creatie van een abstract prototyping (AP) concept. In de opzet van dit concept zijn evaluaties vergeven met het DSS-ontwerpproces. Dit lokaliseert niet alleen fouten tijdens het ontwerp van DSS, maar geeft ook een zekere mate van vertrouwen dat het ontwikkelingsproces zich op het juiste spoor bevindt. Het concept gaat uit van een iteratieve benadering met stapsgewijze verbeteringen, waarbij de beschouwde DSS herhaaldelijk in prototype wordt gezet en geëvalueerd op verschillende abstractieniveaus. Hiermee is bepaald dat theorieën, methoden, algoritmen en proefmodellen testbare in-process implementaties zijn. Deze in-process implementaties worden abstract prototypes genoemd. Eisen spelen de centrale rol in AP in de zin dat zij geclusterd zijn volgens de abstractieniveaus en indirect dienen als evaluatiecriteria. De representanten van de verschillende belanghebbende worden systematisch betrokken bij de evaluatie van de abstract prototypes.

- Ontwikkeling van prototype-softwaregereedschappen die de ontwikkelaars van DSS in AP ondersteunen. Tijdens de eerste stadia van het AP-proces ondersteunt de software voorbereidingsactiviteiten, namelijk (i) verwerving van eisen, (ii) creeren van de representaties van de abstract prototypes, (iii) beschikken over expert begeleiding, bijvoorbeeld tijdens de selectie van representatievormen van de abstract prototypes en bij de selectie van subjecten en (iv) gebruikmaking van de AP-kennis en specifieke uitvoeringsrichtlijnen. Bij de laatste stadia van het AP-proces ondersteunt de software het verwerken van veldinformatie. De AP-software is ook geschikt om eisen, gegevens van abstract prototypes en de analyseresultaten op te slaan voor
gebruik tijdens het DSS-ontwikkelinterval of bij toekomstige projecten. De software vereenvoudigt ook het AP-proces met behulp van specifieke sjablonen en door verbindingen met verscheidene informatiebronnen aan te bieden.

- Practische toepassing van de AP-methodologie en de AP-software in het ontwikkelingsproces en bij het evalueren van de abstract prototypes van DSS. De meeste ontwikkelaars en anderen die AP hebben toegepast oordeelden unaniem dat het een natuurlijke en logische manier van ontwerpen is en van het zeker stellen van kwaliteit en betrouwbaarheid van DSS. Case study's hebben aangetoond dat AP (i) een stevige kader biedt voor het vormen en richten van DSS-ontwerpactiviteiten, (ii) een systematische methode biedt om verschillende belanghebbenden te betrekken bij het progressief controleren of aan de eisen wordt voldaan en (iii) de ontwikkelaars voorziet van middelen om te kunnen meten hoe goed de abstract prototypes zijn ontwikkeld. Met maatstaven en metingen kunnen oplossingen systematisch worden geselecteerd, kunnen zwakke plekken in de abstract prototypes worden gedefinieerd en kan vroegtijdig gesignaleerd worden of er behoefte is aan uitbreiding. In vergelijking met requirements engineering, kan over AP worden gesteld dat het (a) de verwerving van eisen vergemakkelijkt en dat het (b) de middelen verschafte om te meten hoe goed aan de eisen is voldaan en ook om de prioriteiten vast te stellen die aan de verschillende eisen moeten worden gegeven.

De belangrijkste verworvenheden van dit onderzoekswerk kunnen worden samengevat als de bekendmaking en rechtvaardiging dat ontwikkelaars werken met DSS in de ontwerpfase in vier verschillende gedaantes, namelijk theorieën, methoden, algoritmen of proefmodellen, en creatie van:

- een theorie en een computer-gebaseerde methodologie voor pre-implementatie testen van DSS;
- een model voor het richten en vormen van activiteiten in de DSS ontwerp Lane;
- een systematische procedure om verschillende belanghebbenden in het ontwerp van de DSS te betrekken; en
- een kader voor het classificeren en testen of aan de eisen wordt voldaan terwijl DSS wordt ontworpen, overwegend zonder de noodzaak van het schrijven van codes.

Er kan worden gesteld dat AP de gevoeligheid voor fouten van de abstract prototypes reduceert en voorkomt dat slechte beslissingen die anders het DSS project in gevaar brengen of de kwaliteit en betrouwbaarheid ongunstig beïnvloeden van de uiteindelijke DSS, worden geïnstauratied. Het bevorderd al vroeg om doelstellingen, mogelijkheden en verschillende kwaliteitskarakteristieken op een systematische wijze te beschouwen en het helpt de onvoldoeningen te reduceren die vaak leiden tot nabewerkingen wanneer een DSS is afgekeurd. Een resultaat van toepassing van dit methodologie is dat bewezen en betrouwbare theorieën, methoden en algoritmes kunnen worden gebruikt bij de creatie van DSS. De toepassing van deze methodologie kan uiteindelijk de realisatie van bruikbare DSS met laag risico garanderen.
Biography

Eliab Z. Opiyo attended primary, secondary and high schools in Tanzania. He earned a Bachelor of Science degree in Mechanical Engineering from University of Dar es Salaam, Tanzania in 1988 and a Master of Science degree in Computer Aided Design and Manufacturing from Cranfield University, United Kingdom in 1992. From 1992 to 1998 he worked as an academic staff at the University of Dar es Salaam and he was actively researching in the area of computer aided design and manufacturing (CAD/CAM), focusing on the development of tool management systems and building software tools to aid capturing process planning information when designing. During this period, he was also involved in teaching undergraduate Mechanical Engineering courses, namely, production engineering, CAD/CAM, engineering mechanics, and dynamics. He has also been a visiting research associate for a short while at the School of Metallurgy/Inter-disciplinary Research Center (IRC), University of Birmingham, in the United Kingdom in 1996, and a visiting research fellow at the Production Engineering Laboratory, Faculty of Mechanical Engineering and Marine Technology, Delft University of Technology, in the Netherlands from November 1997 to July 1998. In September 1998 he joined Delft University of Technology as a Research Assistant (i.e. Assistent in Opleiding - AIO) and Ph.D. Student at the Faculty of Industrial Design Engineering. His industrial experience include working as a student engineer in Tanzania with the following firms: the Small Industries Development Organization, where he worked at the machine tools and assembly departments; Mwanza Tannersies Company Ltd., where he worked at the maintenance department; and the National Milling Cooperation Ltd., where he worked at the design department. His research interests are in engineering software development, CAD/CAE/CAM, manufacturing processes, tool management, process planning and requirements engineering; and he has published over 20 journal and conference papers.
Index

Reference model, 43-46
Scenarios, 122-123
Software, 147-160
  Activities supported by, 82
  Allocation of features, 81-82
  Application of specific Tools, 194-196
  Architectural design, 85-87
  Associated software, 148
  Auxiliary utilities, 102-104
  Database, 99
  Design, 78
  Hardware, 148
  Hypertext search tool, 104
  Implementation of, 88-105
  Knowledgebase, 101, 147-148
  Main elements, 147
  MDKS utility, 92
  MDRS utility, 82
  Needs analysis, 81
  On-line mentor, 104
  Operating system, 148

Abstract prototype(s), 49
  Authoring of, 91
  Authoring session, 151
  Browsing, 92
  Forms of representation of, 76
  Update of, 92
  Relationships among, 62-64
  Tracking, 77-78
Abstract prototyping, 47
Activities, 65
  Computer-based, 72
  Chronology of, 67
  Front end, 67
  Downstream, 68
Application scenarios, 71
Basic terms, 46-47, 165-167
Deployment scenarios, 130
Driving schemes, 69
  Intra-abstraction level, 69
  Stakeholders involvement, 69
Formal methods, 132
Information flow, 87
Process, 65-80
I/O, 51
Requirements for, 82-84
RRP utility, 85, 89, 194
KRP utility, 85-86, 195
Using the, 108-110
Abstraction, 28
Abstraction levels, 49-51
Acceptability
  Function, 173
  Index, 79, 95-96, 172
Analysis
  Matrix, 171
  Procedure, 171

Capability maturity model, 24-25
Case Studies, 107-128, 175-196
  Industrial, 111-112, 116-122
  Lessons learnt, 125
  Methods used, 107-108
  Research used, 112-116
Confidence
  Function, 173
  Index, 79, 97, 172
Contextual inquiry, 32

Design Support Software
  Analysis of the processes of
  Development, 16-19
  Development of, 5
  Disembodiment of, 44
  Quality assurance of, 6, 15-42
  Scheme for development of, 17

Developers
  Roles of, 51-53
  Knowledge of, 33

Feature based decomposition theory, 182-183
Fulfillment diagram, 174

Handling
  Comments, 123
  Suggestions, 123
Highly interactive communication, 9, 178
Human computer interaction, 179

ICA research program, 8
Information gathering matrix, 169
  Creation of, 93
Information analysis, 78-80, 95, 153, 171

Knowledge
  Acquisition of, 75-76
  Management of, 61
  Maturity of, 61
  Multi-dimensional knowledge search (MDKS), 92
  Processing, 75-76
  Representation of, 62, 75-76

Measurements, 36-37, 169-174
Metrics, 36-37, 169-174
MicroScribe, 115

N

Natural internal representation, 179
Non-computerized operations, 148

O

Opinions gathering, 70
   Instruction for, 94
   Matrix, 169
   Module, 93
   Tool, 169
   Utility, 86

P

Participatory design, 32
Particle system theory, 182
Photogrammetry, 189-194
Physical concept modeling, 179
Product development, 1-3
   Activities, 2

Q

Quality assurance
   User centered, 32
   of DSS, 6, 15-42

R

Relevance
   Function, 173
   Index, 79, 96, 172
Requirements
   Acquisition of, 59, 175
   Algorithm for ranking of, 97
   Browsing, 91
Characteristics of, 54, 161-162
Classification of, 56
Clustering of, 73-73
Dependencies of, 54
Editing, 150
Engineering, 30-32, 53
Functional, 56
Generic, 56
Input, 150
Management of, 161-163
Matching problem to, 98-99
Multi-dimensional requirements search (MDRS), 89-90
Non-functional, 56
Pool of, 60
Prioritization function, 97
Processing of, 73-74, 89, 152
Protocols for acquisition of, 161-163
Quality, 56
Ranking, 97
Relevance of, 57-58
Representation of, 73-74, 89
Sources of, 59
Specific, 56
Validation of, 7-8
What to observe when formulating, 162-163
Requirements model, 59-60
Preparation of, 74-75
Research
   Activities, 12
   Background, 1
   Contributions of, 136-137
   Findings of, 129-136
   Hypothesis, 11, 128
   Implications of, 129-136
   Limitations, 137-138
   Primary research, 8-10
Problem, 10-11
Publications, 13

Review
Materials, 93
Methods, 28
of Algorithms, 186
of Methods, 183, 192
of Pilot prototypes, 187
of Theories, 182

Review Criteria, 47
Levels of fulfillment, 95
Preparation of, 74-75
Validity of, 58-59

S
Shape conceptualization, 8
Shape reengineering, 111, 116

Software
Automated software production, 23-24
Process management concepts, 24-25
Prototyping, 29
    Evolutionary, 29
    High fidelity, 30
    Horizontal, 29
    Low fidelity, 30
    Modular, 29
    Rapid, 29
    Vertical, 29
Testing, 26
Validation, 25-26
Verification, 25-26

Software process models, 19
    Component reuse, 20, 88
    Early-phase oriented models, 21
    Evolutionary model, 21
    Incremental models, 20
    Linear models, 20
    Operational model, 21
    Program growth, 20
    Rapid application, 22
    Rapid prototyping, 22
    Recursive models, 21
    Spiral model, 21
    V model, 20
    Waterfall, 20

Software quality standards, 34-36
    de facto, 34-35
    de jure, 36
    IEEE, 35
    ISO, 35
    ISO/IEC, 35, 60
    Official, 35
    PASs, 35-36

Speech recognition, 114-115
Subjects
    Selection of, 170
    Suitability diagram, 174

U
User interface(s), 104, 148
Users
    Role of, 51-53

V
Vague modeling, 113, 114
Vagueness, 180

W
Warehouse, 99, 147-148
Weak spot analysis, 174
WIMP, 4
Facilitating the Development of Design Support Software by Abstract Prototyping

Many developers of software processes recognize that abstract techniques cannot precisely satisfy the needs in the development of software tools used in engineering design (variously known as CAD, CAFE, or CAD/CAM systems), and introduces a novel quality assurance strategy called abstract prototyping (AP). AP is in a sense a preshadowing implementation review procedure, dedicated to the early development phases. It helps to ensure that the right foundational theories, methods, and algorithms are deployed, and in addition guides the participation of stakeholders in the development process.

This publication first reviews quality assurance strategies used in software development and then introduces the AP concept. Afterwards, it describes the AP process and software tools built to support the developers in doing the AP activities. It also presents case studies conducted to explore the validity of the AP methodology.