ACKNOWLEDGMENTS

The author would like to thank Nanyang Technological University and U.S. Air Force Research Laboratory for funding this research, and the following people for their significant contribution:

Professor Martin G. Helander who supervised and facilitated this research by providing invaluable guidance and suggestions, and also creating plenty of opportunity for independent thinking.

Mr Lo Shuan who provided literature, comments, and recommendations related to DESA and inspection methods.

Mr Tadwalkar Yogesh Anant and Mr Kow Yong Ming who provided insights of UI evaluation in the UI design field.

The author would also like to express gratitude to fellow students in the Centre of Human Factors and Ergonomics for exchanging research ideas and commenting on this research.
TABLE OF CONTENTS

ACKNOWLEDGMENTS ........................................................................................................1
TABLE OF CONTENTS ........................................................................................................2
LIST OF FIGURES ...........................................................................................................5
LIST OF TABLES .............................................................................................................7
SUMMARY ..........................................................................................................................8

CHAPTER 1. Introduction ...................................................................................................9
  1.1 The Science of Design ................................................................................................9
  1.2 Classification of Formal Design Methods .................................................................13
    1.2.1 Classifying Formal Design Methods into Product Development Stages ..........14
    1.2.2 Classifying Formal Design Methods into Elements of Quality .......................17
  1.3 Objectives and structure of thesis ............................................................................20

CHAPTER 2. Literature Review ..........................................................................................22
  2.1 AD (Axiomatic Design) ............................................................................................22
  2.2 DESA (Design Equations for Systems Analysis) .....................................................26
  2.3 DSM (Design Structure Matrix) ..............................................................................29
  2.4 Complementing DSM (Design Structure Matrix) with AD (Axiomatic Design) ..32

CHAPTER 3. Case Study: Differing Roles of AD and DSM in Reducing System Complexity 34
  3.1 Introduction of Case Study ......................................................................................34
  3.2 Employing DSM to Reduce the Complexity of a Hypothetical Water Faucet System 34
  3.3 Employing AD prior to DSM to Reduce the Complexity of a Hypothetical Water Faucet System ..........................................................36
  3.4 Conclusion of Case Study .......................................................................................40

CHAPTER 4. Revising DESA .............................................................................................41
  4.1 Shortcomings of DESA (Design Equations for Systems Analysis) .......................41
  4.2 Revised DESA Framework ......................................................................................42
  4.3 Establishing a DESA Terminology .........................................................................44
  4.4 Examples of Product Planning and Concept Development using the DESA Framework ..........................................................47
    4.4.1 Usability Design of Water Faucet ...................................................................47
    4.4.2 Usability Design of Microscope Workstation .................................................48

CHAPTER 5. Case Study: Using DESA to Design a More Usable Inspection Method ..50
5.1 Introduction of Case Study .................................................................50
5.2 Overview of Usability Inspection Methods .........................................50
5.3 Usability Problems of Inspection Methods ........................................52
5.4 Designing a More Usable Inspection Method ......................................55
  5.4.1 Interviewing UI (user interface) Evaluators to Understand the Context of Use of Inspection Methods ..........................................................55
  5.4.2 Employing DESA to Design the Inspection Method .......................58
5.5 Discussion of Case Study ...................................................................60
5.6 Conclusion of Case Study ..................................................................62

CHAPTER 6. Case Study: Using DESA to Reduce the Complexity of Object-Oriented Software ..............................................................63

6.1 Introduction of Case Study ..................................................................63
6.2 Dependencies in Software Systems .....................................................63
6.3 Object-Oriented Software Design and its Shortcoming .......................64
6.4 Complementing Object-Oriented Design with Axiomatic Design .........65
6.5 Shortcoming of ADOSS (Axiomatic Design of Object-oriented Software Systems) ..........................................................67
  6.6 Complementing Object-Oriented Design with DESA .......................69
6.7 Nim Game Specification using DESA Design Model .........................70
6.8 Dependency Analysis of Nim Game ....................................................76
6.9 Conclusion of Case Study ..................................................................79

CHAPTER 7. Conclusion ...........................................................................81

REFERENCES ..........................................................................................83

APPENDIX A. Formal Design Methods ......................................................91
  A.1 Formal Design Methods for Designing Product Manufacturing Processes ........91
    A.1.1 SPC (Statistical Process Control) ....................................................91
    A.1.2 Acceptance Sampling .................................................................92
    A.1.3 JIT (Just-in-time) .................................................................93
    A.1.4 IDEF (Integrated DEFinition Language) ......................................96
    A.1.5 Reliability Testing .................................................................98
    A.1.6 DFM (Design for Manufacturing) .............................................101
    A.1.7 Six Sigma Process Improvement .............................................103
  A.2 Formal Design Methods for Designing Products ...............................105
    A.2.1 Design Optimization .................................................................105
    A.2.2 Design of Machine Elements ................................................106
    A.2.3 DOE (Design of Experiments) .................................................107
<table>
<thead>
<tr>
<th>A.2.4</th>
<th>Function Analysis Diagram</th>
<th>108</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.2.5</td>
<td>Morphological Chart</td>
<td>109</td>
</tr>
<tr>
<td>A.2.6</td>
<td>SADT (Structured Analysis and Design Technique)</td>
<td>111</td>
</tr>
<tr>
<td>A.2.7</td>
<td>QFD (Quality Function Deployment)</td>
<td>113</td>
</tr>
<tr>
<td>A.2.8</td>
<td>TRIZ (Theory of Inventive Problem Solving)</td>
<td>115</td>
</tr>
<tr>
<td>A.2.9</td>
<td>FMEA (Failure Mode and Effect Analysis)</td>
<td>116</td>
</tr>
<tr>
<td>A.2.10</td>
<td>RD (Robust Design)</td>
<td>117</td>
</tr>
<tr>
<td>A.2.11</td>
<td>DFSS (Design for Six Sigma)</td>
<td>119</td>
</tr>
<tr>
<td>A.2.12</td>
<td>MUSE (Method for Usability Engineering)</td>
<td>120</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1.1. A generic product development process .............................................. 15
Figure 1.2. Structural overview of thesis ............................................................ 21
Figure 2.1. Effects of coupling on design task completion time .......................... 23
Figure 2.2. Design domains used in axiomatic design ......................................... 24
Figure 2.3. Decomposition hierarchies in the functional and physical domains ...... 24
Figure 2.4. DESA framework for analyzing user task designs ............................ 27
Figure 2.5. DSM: clustering of inter-dependent DPs into modules ...................... 29
Figure 2.6. DSM: proto-module B is dependent on proto-module A .................... 31
Figure 2.7. DSM: information hiding modularity ................................................ 31
Figure 3.1. Water faucet with hot and cold water knobs ..................................... 36
Figure 3.2. Water faucet with temperature control functionally isolated from flow rate control ................................................................. 38
Figure 4.1. Revised DESA framework ............................................................... 43
Figure 5.1. Coupled Nielsen's usability heuristics .............................................. 53
Figure 5.2. Less coupled Nielsen's usability heuristics. Only the small matrix in the center is coupled ................................................................. 54
Figure 5.3. Decomposing the functional requirement and the design parameter .... 59
Figure 5.4. Decomposing the ergonomic requirement and the ergonomic parameter 60
Figure 6.1. V model: software design model of ADOSS .................................... 66
Figure 6.2. Design matrix of the Acclaro software (Do and Suh, 2000a) .............. 67
Figure 6.3. Design matrix of the Acclaro software, excluding the user interface .... 68
Figure 6.4. DESA design model ................................................................. 69
Figure 6.5. Screenshots of Nim Game application .............................................. 70
Figure 6.6. UG-FR design matrix of Nim Game ................................................ 76
Figure 6.7. The two off-diagonal ‘X’s imply that DP_{m3.1} and DP_{m3.2} cause an inter-dependency between FR2.1 and FR2.2 ............................................. 77
Figure 6.8. Absence of off-diagonal ‘X’ implies that DP_{m3.1} and DP_{m3.2} do not cause any dependency between FR2.1 and FR2.2 ............................................. 77
Figure 6.9. The two off-diagonal ‘X’s imply that DP_{u3.1} and DP_{u3.2} cause an inter-dependency between FR3.1 and FR3.2 ............................................. 78
DESA: A Design Method that Reduces System Complexity

Figure 6.10. Absence of off-diagonal ‘X’s imply that DPui 3.1 and DPui 3.2 do not cause any
dependency between FR3.1 and FR3.2 .................................................................79

Figure 7.1. Overview of findings and problems solved ..................................................82

Figure A.1. Process out of control and process in control.............................................91
Figure A.2. Control charts for means and ranges.........................................................92
Figure A.3. Single and multiple sampling plans.........................................................94
Figure A.4. Components of production lead time.......................................................95
Figure A.5. Framework for reducing lead time............................................................95
Figure A.6. Basic element of an IDEF0 process map ..................................................96
Figure A.7. First and second level process map of a PCB manufacturing process ..........97
Figure A.8. Weibull plot of data in Table A.1 to obtain β, η, and γ.................................100
Figure A.9. Flowchart of DFM .................................................................................102
Figure A.10. Decomposition of manufacturing costs......................................................103
Figure A.11. Marginal achievement of Six Sigma goal – products that are plus and minus six
standard deviations from the mean fall exactly on the specification range limits ..........104
Figure A.12. Five-stage process: DMAIC.................................................................105
Figure A.13. First-level function analysis diagram.....................................................108
Figure A.14. Second-level function analysis diagram..................................................109
Figure A.15. First-level and second-level of the SADT Activity Diagram....................113
Figure A.16. Customer attributes, CTSs, functional requirements, design parameters, and
process variables in QFD.........................................................................................113
Figure A.17. "House of quality" of customer requirements versus design parameters ....114
Figure A.18. Two designs of a 100m³ cylinder............................................................118
Figure A.19. Two-step optimization process of RD......................................................119
Figure A.20. The structured framework of MUSE.......................................................121
LIST OF TABLES

Table 1.1. The task structure for design (Chandrasekaran, 1990).................................12
Table 1.2. Design methods employed in every product development stage .....................17
Table 1.3. Design methods employed for different elements of quality ........................18
Table 1.4. Definition of each element of quality .........................................................19
Table 5.1. Designing an inspection method using DESA .............................................58
Table 5.2. Eight requirements for a usable inspection method ........................................60
Table 6.1. Decomposed UGs, FRs, DPms, DPuis, and UAs ...........................................74
Table 6.2. Responsibilities of the design parameters .....................................................75

Table A.1. Times-to-failure data and the corresponding median rank ............................98
Table A.2. Offset sidebar chain dimensions ................................................................107
Table A.3. Morphological chart ..................................................................................110
Table A.4. Concept selection matrices ........................................................................111
Table A.5. Design FMEA worksheet ............................................................................117
SUMMARY

This research deals with a formal design method called “Design Equations for Systems Analysis”, or DESA. This method reduces system complexity, and has been demonstrated to effectively reduce the complexity of human-machine systems. It also increases the usability of products. The author summarized two other design methods that reduce system complexity – Axiomatic Design (AD) and Design Structure Matrix (DSM) – and explained their differences from DESA. A case study was then presented to demonstrate the ability of AD and DSM to complement each other. Since DESA builds on AD, it can complement DSM similarly. The author revised the framework that DESA utilizes. A terminology was established to define the technical terms in DESA. DESA was then employed to design an inspection method that evaluates usability. It was also employed to reduce the complexity of an object-oriented software system. These two applications of DESA are presented as case studies. These two case studies respectively demonstrate that DESA designs for usability and effectively reduces the complexity of object-oriented software systems.
CHAPTER 1. Introduction

1.1 The Science of Design

Scientific endeavors have primarily dealt with the natural environment: physics, biology, chemistry, and geology. They were termed as natural sciences, and natural laws were discovered from these studies. Scientific studies have later expanded to include human behavior and human relationships, such as in anthropology, psychology, economics, political science, and sociology. These were termed as social sciences or behavioral sciences. Only a few decades ago, humans saw the need for scientific study of the artificial environment, as much of our world contains man-made artifacts and events (Simon, 1969). All artificial things created – such as engineered products, building architectures, business plans, educational policies, legislatures, and medical procedures – have to be designed, but sometimes without much thought. The science of design is inevitable and is an important part of the sciences of the artificial.

Our forefathers’ primitive approaches are utilized in engineering design, and much of it was based on designers’ intuition, experience, and conventional wisdom. Nevertheless, these characteristics are now complemented, as studies were made over the past few decades to transform engineering design from an art to a science. Simon (1969) suggested that in addition to natural science, which is knowledge about natural objects and phenomena, there should also be a science of the artificial for knowledge about artificial objects and phenomena. He demonstrated the existence of a number of components of a theory of design, and the existence of a substantial amount of theoretical and empirical knowledge of it. These include:

1. The evaluation of designs
   a. Theory of evaluation: utility theory, statistical decision theory
   b. Computational methods
      i. Algorithms for choosing optimal alternatives such as linear programming computations, control theory, dynamic programming
ii. Algorithms and heuristics for choosing satisfactory alternatives

2. The formal logic of design: imperative and declarative logics

3. The search for alternatives
   a. Heuristic search: factorization and means-ends analysis
   b. Allocation of resources for search

4. Theory of structure and design organization: hierarchic systems

5. Representation of design problems

The theory of evaluation presented in the above listing aids systematic analysis of the logics that designers used in their designs. Computational methods are tools that aid the selection of an optimal or satisfactory design solution among various alternatives, taking into consideration of the inner constraints and outer environmental parameters.

In formal logic of design, Simon highlighted the paradoxes of imperative logic and explained that employing a special system of imperative logic to handle the paradoxes is unnecessary, as an adaptation of the ordinary declarative logic is adequate for the requirements of design to be fully met. The search for alternatives section introduces heuristic-based tools that employ a logical search process to identify a satisfactory alternative. Theory of structure and design organization explains that complex systems have hierarchical structure whereby several components in the structure perform different sub-functions, and they work together to fulfill an overall function. A powerful technique to design a complex system is to decompose it into its components, in relation to their corresponding functions, and examine them separately. Lastly, representation of design problems highlights that a systematic and appropriate representation of design problems would significantly simplify the problem solving process.

Ross and Schoman (1977) presented comprehensive concepts for the definition of design requirements, which included context analysis, functional specification, and design constraints. Based on these concepts, Ross and Schoman developed a systematic design method termed “Structured Analysis and Design Technique” (SADT). A summary of this method is presented in Appendix A.2.6.
Chandrasekaran (1990) analyzed a general class of design methods termed as propose-critique-modify methods, and then proposed a design task structure by identifying a range of methods for each design task (Table 1.1).

Suh (1990) proposed two design axioms termed the “independence axiom” and the “information axiom”, and derived design corollaries and design theorems based on these two axioms. The independence axiom states: maintain the independence of functional requirements. The information axiom states: minimize the information content of the design. From the dictionary, the word “axiom” has two meanings: an established or accepted principle, or a self-evident truth (Thompson, 1993). The latter meaning is more common in ancient philosophy, when an axiom was a starting point that needed no evidence or demonstration, because it was assumed to be self-evident and necessarily true (Popkin, 1991).

Suh’s definition of axiom takes the former meaning of the dictionary, because the design axioms are accepted design principles based on systematic observations, not self-evident truth. Similar to natural laws, these design axioms are unproven, and also unprovable, but they are assumed to be valid as long as no counter-examples that violate them can be found. While natural laws give precise descriptions, explanations, generalizations, or predictions to regular patterns that occur in nature, design axioms give the same precise expressions to regular patterns that occur in good designs. Design axioms, in addition to natural laws, will end more meaningless arguments over the merits of different designs (Brown, 2006) – besides using physical laws to objectively evaluate efficiency, stability, durability, and strength of a designed artifact, designers can use design axioms to objectively evaluate its effectiveness, reliability, manufacturability, productivity, and usability.
Table 1.1. The task structure for design (Chandrasekaran, 1990)

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Methods</th>
<th>Subtasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design</td>
<td>Propose, critique, modify family (PCM)</td>
<td>Propose, verify, critique, modify</td>
</tr>
<tr>
<td>Propose</td>
<td>Decomposition methods (incl. design plans) and transformation methods</td>
<td>Specification generation for subproblems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Solution of subproblems generated by decomposition (another set of design-tasks)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Composition of subproblems solution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Match and retrieve similar case</td>
</tr>
<tr>
<td></td>
<td>Case-based methods</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Global constraint-satisfaction methods</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a. Numerical optimization methods</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b. Numerical or symbolic constraint propagation methods</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specification generation for subproblems</td>
<td>Constraint propagation incl. constraint posting</td>
<td>Simulation to decide how constraints propagate</td>
</tr>
<tr>
<td>Composition of subproblem solutions</td>
<td>Configuration methods</td>
<td>Simulation for prediction behavior of candidate configurations</td>
</tr>
<tr>
<td>Verify</td>
<td>Domain specific calculations or simulation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Qualitative simulation, consolidation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a. Visual simulation</td>
<td></td>
</tr>
<tr>
<td>Critique</td>
<td>Causal behavioral analysis techniques to assign responsibility</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dependency-analysis techniques</td>
<td></td>
</tr>
<tr>
<td>Modify</td>
<td>Hill-climbing-like methods which incrementally improve parameters</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dependency-based changes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Function-to-structure mapping knowledge</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Add new functions</td>
<td>Design new function. Recompose with candidate design</td>
</tr>
</tbody>
</table>
1.2 Classification of Formal Design Methods

Based on design concepts and axioms, many formal design methods have been developed to guide product developers to design objectively, systematically, and easily, and enable them to present their design plans clearly.

Webster’s Dictionary defines formal as definite, orderly, and methodical. It defines method as a regular, orderly, and definite procedure. Hence, a formal design method should have a methodical usage procedure, or a structured framework, or both. However, a formal method does not necessarily require the use of formal logic, or even mathematics (Luqi and Goguen, 1997). Therefore, design concepts and guidelines, such as Poka-yoke, Total Design, and Heuristic Evaluation, are not considered formal design methods in this research.

Below are listed several examples of state-of-the-art formal design methods in the product development field. Summaries of these methods can be found in Appendix A.

2. AD (Axiomatic Design) – Suh, 1990
3. Design of Machine Elements – Shigley et. al., 2004
5. DFM (Design for Manufacturing) – Ulrich and Eppinger, 2000
7. DOE (Design of Experiments) – Condra, 2001
8. DSM (Design Structure Matrix) – Baldwin and Clark, 2000
9. FMEA (Failure Mode and Effect Analysis) – Yang and El-Haik, 2003
10. Function Analysis Diagram
12. JIT (Just-in-time) – Monden, 1993
13. Morphological Chart
15. QFD (Quality Function Deployment) – Franceschini, 2002
16. RD (Robust Design) – Fowlkes and Creveling, 1995
The subsequent sections show two different classification of DESA (Design Equations for Systems Analysis) with these formal design methods. They reveal the role of DESA as a formal design method in the product development field.

1.2.1 Classifying Formal Design Methods into Product Development Stages

A generic product development process has six stages (Ulrich and Eppinger, 2000):

1. Planning
2. Concept development
3. System-level design
4. Detail design
5. Testing and refinement
6. Production ramp-up

The first four stages constitute the product design process, and the last two stages constitute the product manufacturing process (Figure 1.1). In the planning stage, design problems are defined by specifying the product requirements and identifying the design constraints. Design parameters that satisfy the requirements and constraints are then conceptualized at the concept development stage. During system-level design, components within the product's complex systems are integrated, and the interfaces and interactions between them are analyzed at a system-level. At the detail design stage, the focus is to gather all the necessary information for manufacturing. Typical tasks include detailed calculation of forces, dimensioning, materials selection, and drawings. Another common set of terms for these four stages is product definition, conceptual design, embodiment design, and detail design. At the product
manufacturing stages, the product is tested and refined before ramping-up the production at the final stage.

Many formal design methods have been developed to describe various stages of the product development process (Chen, 1999). Methods in the first four stages are employed to design the product. Methods in the last two stages are employed to design the manufacturing process with the aim of increasing productivity and product consistency. Therefore, the formal design methods listed in the preceding section can be segregated into two groups:

1. Formal design methods for designing products:
   - AD (Axiomatic Design)
   - DESA (Design Equations for Systems Analysis)
   - Design of Machine Elements
   - Design Optimization
   - DFSS (Design for Six Sigma)
   - DOE (Design of Experiments)
   - DSM (Design Structure Matrices)
   - FMEA (Failure Mode and Effect Analysis)
   - Function Analysis Diagram
   - Morphological Chart
   - MUSE (Method for Usability Engineering)
DESA: A Design Method that Reduces System Complexity

- QFD (Quality Function Deployment)
- RD (Robust Design)
- SADT (Structured Analysis and Design Technique)
- TRIZ (Theory of Inventive Problem Solving)

2. Formal design methods for designing product manufacturing processes:
- Acceptance Sampling
- DFM (Design for Manufacturing)
- IDEF (Integrated DEFinition Language)
- JIT (Just-in-time)
- Reliability Testing
- Six Sigma Process Improvement
- SPC (Statistical Process Control)

Among the methods listed above, DSM is the only method that also designs the product design process (Baldwin and Clark, 2000). Methods that design the design process are employed to manage personnel involved in a product development project and their tasks. However, methods that solely design the product design process are excluded in this research. TQM (Total Quality Management) is an example of such methods.

Table 1.2 classifies the methods into the six distinct stages of the product development process. There is no lack of design methods in each product development stage. In the first three stages, where DESA is employed (Lo and Helander, 2004b), many other design methods can be found.

However, design methods between product development stages are not equally established – the first few stages are much less established than the last few stages. This is because initial efforts of product quality improvement target the last few stages of product development. The earliest developed design methods focused on the design of manufacturing systems to achieve consistent products. The third and fourth stages contain established families of methods such as Design Optimization and Design of Machine Elements. Ironically, the first two stages, where the impact on the
product is the greatest, are not well-established (Oh, 2006). Products with wrongly specified requirements and poorly conceptualized solutions are difficult to optimize or improve. Correcting a wrong product will not yield customer satisfaction. To maximize satisfaction, it is necessary to develop the right product from the very beginning. DESA contributes to the front-end of the product development process, as it is employed in the first three stages (Lo and Helander, 2004b).

Table 1.2. Design methods employed in every product development stage

<table>
<thead>
<tr>
<th>Product development stages</th>
<th>Design methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Planning</td>
<td>DOE, Function Analysis Diagram, SADT, QFD, TRIZ, DFSS, MUSE, AD, DESA</td>
</tr>
<tr>
<td>2. Concept development</td>
<td>DOE, Morphological Chart, SADT, QFD, TRIZ, FMEA, RD, DFSS, MUSE, AD, DESA</td>
</tr>
<tr>
<td>4. Detail design</td>
<td>Design Optimization, Design of Machine Elements, DOE, RD, DFSS</td>
</tr>
<tr>
<td>5. Testing and refinement</td>
<td>SPC, Acceptance Sampling, JIT, IDEF, Reliability Testing, DFX, Six Sigma Process Improvement</td>
</tr>
<tr>
<td>6. Production ramp-up</td>
<td>SPC, Acceptance Sampling, JIT, DFM, Six Sigma Process Improvement</td>
</tr>
</tbody>
</table>

1.2.2 Classifying Formal Design Methods into Elements of Quality

Formal design methods, developed for various stages of the product development process, have a common objective which is to develop a good product. This objective is commonly known as quality assurance. The term “quality” comprises many elements such as consistency, cost-effectiveness, effectiveness, efficiency, functionality, manufacturability, reliability, robustness, safety, simplicity, stability, strength, usability, and workability. Hence, design methods belonging to similar product development stages can be further classified into the elements of quality they design for (Bralla, 1996). For example, although RD (Robust Design) and DESA are both employed in the solution conceptualization stage, RD designs for robustness
DES A: A Design Method that Reduces System Complexity

(Fowlkes and Creveling, 1995), whereas DESA designs for simplicity and usability (Lo and Helander, 2004b).

Table 1.3 classifies the design methods into the various elements of quality. The elements are defined in the context of this study, as shown in Table 1.4.

<table>
<thead>
<tr>
<th>Elements of quality</th>
<th>Design methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistency</td>
<td>SPC, Six Sigma Process Improvement, DFSS</td>
</tr>
<tr>
<td>Cost-effectiveness</td>
<td>JIT, DFM, Design Optimization, QFD, TRIZ, DFSS</td>
</tr>
<tr>
<td>Effectiveness</td>
<td>SADT, QFD, DFSS, MUSE, AD, DESA</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Design Optimization</td>
</tr>
<tr>
<td>Functionality</td>
<td>Function Analysis Diagram, SADT, QFD, TRIZ, DFSS, MUSE, AD, DESA</td>
</tr>
<tr>
<td>Manufacturability</td>
<td>IDEF, DFM, AD</td>
</tr>
<tr>
<td>Reliability</td>
<td>Reliability Testing</td>
</tr>
<tr>
<td>Robustness</td>
<td>RD, DFSS</td>
</tr>
<tr>
<td>Safety</td>
<td>FMEA</td>
</tr>
<tr>
<td>Simplicity</td>
<td>TRIZ, AD, DSM, DESA</td>
</tr>
<tr>
<td>Stability</td>
<td>Design of Machine Elements</td>
</tr>
<tr>
<td>Strength</td>
<td>Design of Machine Elements</td>
</tr>
<tr>
<td>Workability</td>
<td>FMEA</td>
</tr>
<tr>
<td>Usability</td>
<td>MUSE, DESA</td>
</tr>
</tbody>
</table>

Elements of quality, such as functionality, effectiveness, and cost-effectiveness, are dealt with by many design methods. Although elements of quality, such as consistency, efficiency, stability, and strength, do not have many design methods, their design methods are relatively more established. Elements that need more attention are those which contain only a few methods and which the methods are relatively new. Simplicity is one such element. In the context of this study, simplicity is defined as independency between system components or functions, or both, to
reduce system complexity. There are only three state-of-the-art methods that design for simplicity, and the methods are relatively new:

1. AD (Axiomatic Design) – Suh, 1990
2. DSM (Design Structure Matrix) – Baldwin and Clark, 2000
3. TRIZ (Theory of Inventive Problem Solving) – Terninko et. al., 1996

Since DESA also designs for simplicity (Lo and Helander, 2004b), it contributes to the simplicity element of product quality.

Table 1.4. Definition of each element of quality

<table>
<thead>
<tr>
<th>Elements of quality</th>
<th>Definition of each element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistency</td>
<td>Minimal product variation</td>
</tr>
<tr>
<td>Cost-effectiveness</td>
<td>Minimal cost of production, and maximal utilization of resource</td>
</tr>
<tr>
<td>Effectiveness</td>
<td>Ability to fulfill all customer requirements</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Ability to maximize performance with minimal energy used</td>
</tr>
<tr>
<td>Functionality</td>
<td>Exhaustiveness of product functions</td>
</tr>
<tr>
<td>Manufacturability</td>
<td>Ease of production</td>
</tr>
<tr>
<td>Reliability</td>
<td>Probability of non-failure within a specified period of time</td>
</tr>
<tr>
<td>Robustness</td>
<td>Insensitivity to variation of noise</td>
</tr>
<tr>
<td>Safety</td>
<td>Ability to prevent accidents</td>
</tr>
<tr>
<td>Simplicity</td>
<td>Independency between components or functions or both</td>
</tr>
<tr>
<td>Stability</td>
<td>Ability to physically remain firm and steady</td>
</tr>
<tr>
<td>Strength</td>
<td>Ability to withstand internal and external forces and weights</td>
</tr>
<tr>
<td>Workability</td>
<td>Ability to function correctly according to specification</td>
</tr>
<tr>
<td>Usability</td>
<td>Ease of use, usefulness, pleasantness, and ergonomics</td>
</tr>
</tbody>
</table>

Among the methods that design for simplicity, AD, DSM, and DESA have similarities, as all of them utilize matrices to represent dependencies within systems. Nevertheless, there are several differences between them. The following chapter
DESA: A Design Method that Reduces System Complexity

reviews these methods to explain their similarities and differences. The other formal design methods are summarized separately in Appendix A.

1.3 Objectives and structure of thesis

This thesis has five objectives:

1. Demonstrate using a case study the similarities and differences between Axiomatic Design (AD) and Design Structure Matrix (DSM), and show how they can complement each other
2. Explain how Design Equations for Systems Analysis (DESA) differs from AD and DSM
3. Revise the DESA framework to align it with ISO 9241.11, and establish a terminology to distinguish the terms used in the revised DESA framework
4. Demonstrate using a case study how DESA can improve the usability of inspection methods
5. Demonstrate using a case study how DESA can effectively reduce the relational complexity of an object-oriented software system

There are six subsequent chapters in this thesis. The next chapter, Chapter 2, summarizes DESA and two other design methods that reduce system complexity – Axiomatic Design (AD) and Design Structure Matrix (DSM). Differences between the two methods are explained. Chapter 3 documents a case study that demonstrates the ability of AD and DSM to complement each other. Since DESA builds on AD, it can complement DSM in the same manner. Chapter 4 presents the revised framework of DESA and the established terminology. Examples are provided to demonstrate how DESA can be utilized for product planning and concept development. Chapter 5 presents the case study where DESA was employed to design a more usable inspection method. Chapter 6 presents the case study where DESA was employed to reduce the complexity of object-oriented software systems. Finally, Chapter 7 concludes this research. A structural overview of this thesis is shown in Figure 1.2.
Chapter 1: Introduction
Provides an overview of the science of design and classifies state-of-the-art design methods; DESA reduces system complexity and also increases product usability.

Chapter 2: Literature Review
Summarizes DESA and two other design methods that reduce system complexity – Axiomatic Design (AD) and Design Structure Matrix (DSM); compares the similarities and differences between these methods.

Chapter 3: Case Study
Demonstrates the ability of AD and DSM to complement each other; since DESA builds on AD, it can complement DSM similarly.

Chapter 4: Revising DESA
Presents the revised framework of DESA and the established DESA terminology; examples are provided to demonstrate how DESA can be utilized for product planning and concept development.

Chapter 5: Case Study
Presents the case study where DESA was employed to design a more usable inspection method; demonstrates the ability of DESA to increase the usability of products.

Chapter 6: Case Study
Presents the case study where DESA was employed to reduce the complexity of object-oriented software systems; demonstrates the ability of DESA to reduce the complexity of systems.

Chapter 7: Conclusion
Summarizes the findings of this research.

Appendix A
Summarizes the various state-of-the-art formal design methods introduced in Chapter 1.

Figure 1.2. Structural overview of thesis
CHAPTER 2. Literature Review

Complex systems are difficult to build, maintain, and modify due to the dependencies between their numerous components. Nevertheless, this complexity can be reduced if the systems are appropriately designed. DESA (Design Equations for Systems Analysis) is a design method that has the capability to reduce the complexity of systems. Currently, there are two other design methods that reduce system complexity: Axiomatic Design (AD) (Suh, 1990) and Design Structure Matrix (DSM) (Baldwin and Clark, 2000). The following sections review AD, DESA, and DSM.

2.1 AD (Axiomatic Design)

Generally, a system is described as complex when one has problems understanding or dealing with it (Miller, 2000). Previous research largely agrees that there are two main sources of complexity: a large number of parts and a large number of relations (Woods, 1988; Yates, 1978). Miller named these two types of complexity “component complexity” and “relational complexity”, respectively. Suh (2005), however, noted that a system with a large number of parts does not necessarily mean that it is complex. Complexity depends on the number of relations.

Component and relational complexities are both related to the limits of cognitive capacity. People can attend to and mentally manipulate only four independent pieces of information at the same time (Rode, 2000). Therefore, when the number of elements and relations in a system is excessive, people experience great difficulty in predicting the effects of an action, or tracing the implications of a disturbance in a system (Döner, 1996).

An experiment conducted by Hirschi and Frey (2002) investigated the effect of problem scale (component complexity) and coupling (relational complexity) on human capability to carry out parameter design. The results of the experiment showed that the time taken to complete an uncoupled design task increased linearly with the
problem size, but the time taken to complete coupled design tasks increased geometrically with the problem size (Figure 2.1).

![The Effect of Coupling and Scale on Completion Time](image)

Figure 2.1. Effects of coupling on design task completion time

AD provides a method to compute dependencies between system functions (Suh, 1990). This systematizes complexity analysis and hence facilitates complexity reduction. The principal concepts of AD can be summarized as follow:

- Design domains are used to group various attributes in system design (Figure 2.2)
- In a domain, the attributes form a decomposition hierarchy, which is constructed via a "zigzagging" process (Figure 2.3)
- Attributes in one domain are mapped into the adjacent domain such that functional dependencies are minimized
- Design equations are used to represent this mapping process between domains, and they aid the evaluation of functional dependencies
Mapping between FRs and DPs is represented by (Equation 2.1)

\[
\{FR\} = [A] \{DP\} \tag{2.1}
\]

where \(\{FR\}\) is a column vector that contains all the FRs of the design,
\(\{DP\}\) is a column vector that contains all the DPs of the design, and
\([A]\) is the "design matrix" that defines the relationships between the design
parameters and the functional requirements. With an equal number \(n\) of
FRs and DPs, \([A]\) is a square matrix of size \(n \times n\).

Hence, when \(n = 3\), the design matrix is of the following form (Equation 2.2):

\[
[A] = \begin{pmatrix}
A_{11} & A_{12} & A_{13} \\
A_{21} & A_{22} & A_{23} \\
A_{31} & A_{32} & A_{33}
\end{pmatrix} \tag{2.2}
\]
Conventionally, the values of a design matrix will either be 'X' or '0', where 'X' represents a mapping between the corresponding vector components while '0' signifies no mapping.

By examining the design matrix, three levels of functional dependency, or coupling, can be identified:

1. Uncoupled
2. Semi-coupled
3. Coupled

An uncoupled design is one where all non-diagonal elements of the design matrix are zero. There is a one-to-one mapping between FRs and DPs, indicating that the FRs are independent of one another. Such system design with a diagonal matrix is optimal (Equation 2.3).

\[
\begin{bmatrix}
\text{FR1} \\
\text{FR2} \\
\text{FR3}
\end{bmatrix} =
\begin{bmatrix}
X & 0 & 0 \\
0 & X & 0 \\
0 & 0 & X
\end{bmatrix}
\begin{bmatrix}
\text{DP1} \\
\text{DP2} \\
\text{DP3}
\end{bmatrix}
\] (2.3)

However, in a complex real world where interactions of factors are common, an uncoupled design may not be easy to accomplish. Designs where an FR is satisfied by more than one DP can be acceptable, as long as the design matrix is triangular. This is called a semi-coupled design (Equation 2.4). It still satisfies the functional independence criterion, provided that the DPs are specified in a sequence such that one unique DP ultimately controls each FR.

\[
\begin{bmatrix}
\text{FR1} \\
\text{FR2} \\
\text{FR3}
\end{bmatrix} =
\begin{bmatrix}
X & 0 & 0 \\
X & X & 0 \\
X & X & X
\end{bmatrix}
\begin{bmatrix}
\text{DP1} \\
\text{DP2} \\
\text{DP3}
\end{bmatrix}
\] (2.4)
A coupled design is one in which ‘X’s are found on both sides of the matrix diagonal, indicating inter-dependencies between the FRs (Equation 2.5). Such system design is highly complex, as DPs affect one another in both directions. Much iteration is required when the DPs are implemented or modified.

\[
\begin{bmatrix}
\text{FR1} \\
\text{FR2} \\
\text{FR3}
\end{bmatrix}
= \begin{bmatrix}
X & 0 & X \\
X & X & X \\
0 & X & X
\end{bmatrix}
\begin{bmatrix}
\text{DP1} \\
\text{DP2} \\
\text{DP3}
\end{bmatrix}
\tag{2.5}
\]

Among numerous formal design methods, AD is the only method that use design axioms to objectively evaluate design solutions (Section 1.1). Yien and Tseng (1996) reviewed formal design methods that aid solution conceptualization. Their purpose was to find an appropriate method for solution conceptualization of manufacturing systems. AD was selected after the review. Most recently, AD was included in the group of design methods that DFSS (Design for Six Sigma) employs. AD has been demonstrated to effectively reduce the complexity of mechanical systems (Suh, 1990) and software systems (Do and Suh, 2000a; Suh, 2001).

2.2 DESA (Design Equations for Systems Analysis)

Based on AD (Axiomatic Design), Lo and Helander (2004b) proposed a framework that analyzes couplings between the goals that a user wants to achieve and the control actions that are designed into a user interface. This type of coupling increases the gulf of execution (Norman, 1988). Lo and Helander termed the framework as DESA (Design Equations for Systems Analysis), and demonstrated how it can be used to analyze consumer products such as film cameras.

There are four design domains in DESA (Figure 2.4). Each domain contains a unique set of design attributes. A variable-illumination ceiling lamp can be used to illustrate the characteristics of these attributes. The goal domain contains user goals (UGs) that
describe a user’s desired state of a system, such as “appropriate amount of light”. The functional domain contains functional requirements (FRs), which characterize the function of a system, such as “provide a range of illumination”. The physical domain contains design parameters (DPs), which are physical embodiments or variables that are selected by a designer to satisfy the FRs, such as “electrical resistance”. The action domain contains user actions (UAs) that are designed into a user interface for controlling the DPs, such as “rotate lamp switch”.

For each pair of design domains in Figure 2.4, the domain on the left describes the ends; the domain on the right describes the means. The mapping from the goal domain to the functional domain represents decision-making regarding the functions of a system. The mapping from the functional domain to the physical domain represents engineering design of internal structure. The mapping from the physical domain to the action domain represents decision-making regarding the design of user interface.

The goal and action domains form a functional model; they describe control tasks from a user’s perspective. The functional and physical domains form a structural model; they describe the structure of a system from an engineering perspective. To understand and correct usability problems, it is sometimes necessary to analyze the
DESA: A Design Method that Reduces System Complexity

functional model together with the structural model. This is because design decisions in either of the two models constrain the design options that are available in the other (Lee and Yoon, 2004).

Similar to AD, design equations and design matrices are used to represent the mappings between domains, which aid coupling analysis of the systems. However, the domains are now user-centered, and there are three sets of equations and matrices to represent the mapping between UGs and FRs, between FRs and DPs, and between DPs and UAs (Equations 2.6 to 2.8).

\[
\{\text{UG}\} = [A]\{\text{FR}\} \quad (2.6)
\]

\[
\{\text{FR}\} = [B]\{\text{DP}\} \quad (2.7)
\]

\[
\{\text{DP}\} = [C]\{\text{UA}\} \quad (2.8)
\]

Hence,

\[
\{\text{UG}\} = [U]\{\text{UA}\}, \text{ where } [U] = [A][B][C] \quad (2.9)
\]

Equation 2.9 reflects the notion that the directness of human control tasks does not solely depend on user interface design; it also depends on the functional specification and the underlying workings of the system. DESA has been demonstrated to effectively reduce the complexity of human-machine systems, as it analyzes both their underlying workings and their human-machine interfaces, to resolve any couplings (Helander, 2007; Lo and Helander, 2007) – several human factors design problems were analyzed: a refrigerator, hand tools, driver’s compartment, microscope workstation, manual cameras, and a process control micro-world.
2.3 DSM (Design Structure Matrix)

DSM is a tool that manages dependencies between system DPs (design parameters). It was developed by Steward (1981) and refined by Eppinger (1997) to manage large-scale complex systems. The principal concepts of DSM can be summarized as follows:

- DPs of a system are intuitively elicited, often based on experience
- Dependencies between the DPs are represented in matrix form (Figure 2.5)
- An ‘X’ in the matrix indicates a dependency between two DPs – the row-DP depends on the column-DP
- DP2 is concluded to be dependent on DP1, when DP1 is an input to DP2 – modifying DP1 affects DP2
- Modular design (modularity) is achieved by clustering inter-dependent DPs, and by information hiding, to form modules
- Modules are units in a larger system that are structurally independent of one another, but work together (Baldwin and Clark, 2000)

Figure 2.5 shows the clustering of inter-dependent DPs into two modules, as demarcated by the two boxes bounded by the bold lines – DP1 and DP2 constitute module A, while DP3, DP4, and DP5 constitute module B. The characteristic
independency between modules is indicated by the absence of 'X's outside the bounded boxes. However, the modules contain dependencies within themselves, as shown by the 'X's within the two bounded boxes.

Modular design affects the design process in two ways. First, a development team should be assigned to each module – the design structure in Figure 2.5 will hence require two teams: team A and team B. The designers in each team communicate with one another to handle any inter-dependencies within their module. Second, the development teams can work independently from one another, due to the independency between modules – team A’s design tasks do not rely on team B’s and vice versa. Therefore, modular design reduces the complexity of system development. It also reduces the complexity of subsequent system maintenance, modification, and extension – changes can be made to module A without affecting module B and vice versa.

In large-scale complex systems, clustering inter-dependent DPs offers a necessary but not the sufficient condition to achieve modularity. Dependencies are usually present between the clusters – denoted by 'X's outside the bounded boxes (Figure 2.6). This results in a non-modular design. Clusters with dependencies between them are termed "proto-modules" (Baldwin and Clark, 2000; Sullivan et al., 2001). To achieve modularity, information "hiding" has to be performed for the proto-modules. Information hiding was introduced by Parnas (1972) as an approach to develop modular software architecture.

Figure 2.6 shows a modified version of the DSM in Figure 2.5. DP4 is now dependent on DP1, which causes cluster B to be dependent on cluster A. Therefore, the clusters are now proto-modules instead of modules. This results in a non-modular design structure.
Figure 2.6. DSM: proto-module B is dependent on proto-module A

Figure 2.7 shows the resulting DSM after information hiding. The cause of the dependency between DP1 and DP4 was identified and translated into a design rule (DR1) (i.e. design constraint). DP1 and DP4 are now hierarchically dependent on DR1, as indicated by the two 'X's in DR1 column. However, they are now independent of each other. Excluding the 'X's in the DR rows and DR columns, there are no 'X's outside the bounded boxes. Hence, the clusters are now independent of each other, which results in a modular design.

Figure 2.7. DSM: information hiding modularity
For example, if DP1 and DP4 are bottle neck and bottle cap respectively, their dependency would be caused by the need for them to fit at their interface. DR1 will then be the design of the neck-cap interface. This DR has to be set before designing the bottle neck and the bottle cap. However, once the neck-cap interface design is set, the bottle neck and the bottle cap can be designed independently of each other. Thus, information of the bottle neck is hidden from the bottle cap and vice versa, except the information of the neck-cap interface design.

DSM has been demonstrated to effectively reduce the complexity of mechanical systems (Steward, 1981; Eppinger, 1997), computer systems (Baldwin and Clark, 2000), and software systems (Sullivan et al., 2001; Sangal et al., 2005).

2.4 Complementing DSM (Design Structure Matrix) with AD (Axiomatic Design)

AD and DSM may seem to be similar, as both reduce system complexity by utilizing matrices to represent system dependencies. In reality, these two methods are fundamentally different and can complement each other. AD selects appropriate DPs (design parameters) so that system dependencies are minimized. However, an uncoupled design or a semi-coupled design may be unachievable even when the dependencies are minimized. DSM can then be employed to manage the remaining dependencies by manipulating the selected DPs into a modular architecture.

In brief, AD eliminates avoidable dependencies, whereas DSM manages the remaining inherent dependencies – they have different roles in reducing system complexity. Dong and Whitney (2001) demonstrated how an AD design matrix can be converted into a DSM, and argued that such a procedure has 3 benefits:

1. Enable the use of DSM at early stage of the design process, since AD design matrices are constructed at early stage
2. Resulting DSM is driven by FRs (functional requirements) of the system – the DPs of the DSM are systematically translated from the FRs
3. Enable requirements management and system level design knowledge management from early stage of the design process

However, Dong and Whitney (2001) did not explicitly demonstrate how AD complements DSM to effectively reduce system complexity. The following chapter therefore demonstrates, via a hypothetical case study, the different roles of AD and DSM in reducing system complexity. Since DESA builds on AD, it complements DSM in a similar way as demonstrated.
CHAPTER 3. Case Study: Differing Roles of AD and DSM in Reducing System Complexity

3.1 Introduction of Case Study

Large-scale systems are often complex. A major contributing factor of their complexity is the numerous dependencies between their components. Manipulating one component may disrupt other components, and vice versa. This impedes system development, maintenance, and modification. To reduce system complexity, it is essential to minimize and manage these dependencies. Axiomatic design (AD) and Design Structure Matrix (DSM) are methods that have the capability to accomplish this. Apparently, AD and DSM are similar, as both computes dependencies in matrix form to systematize dependency analysis. In reality, they are fundamentally different and have the potential to complement each other. AD minimizes dependencies between system functions. Many of these dependencies are caused by inappropriate system parameters, and can be eliminated by changing the parameters. However, certain dependencies are inevitable, as they are inherent due to the laws of physics and logic. For example, the size and shape of a bottle cap are inherently dependent on the size and shape of its bottle. DSM can then be employed to manage such dependencies by manipulating the system parameters into a modular architecture. In short, AD eliminates avoidable dependencies, whereas DSM manages the remaining inherent dependencies. This study explicitly demonstrates the different roles of AD and DSM in reducing system complexity. We used a mechanical system as a hypothetical case study. AD and DSM were shown to complement each other effectively to reduce system complexity.

3.2 Employing DSM to Reduce the Complexity of a Hypothetical Water Faucet System

In this case study, a hypothetical system design scenario was created. The scenario is explained below.
A system architect was tasked to design an unusual water faucet system. Based on customer requirements, water had to continuously flow from the faucet at a desired temperature, $T_d \ °c$, and desired volume flow rate, $Q_d \ m^3/s$. In addition, the continuously flowing water had to be accumulated in a sink. The architect employed DSM (Design Structure Matrix) to analyze the dependencies in the system. These DPs (design parameters) were identified as the potential candidates that would fulfill the requirements:

$DP_1 =$ Hot water knob  
$DP_2 =$ Cold water knob  
$DP_3 =$ Drain plug

Based on the DPs, the DSM was constructed:

\[
\begin{array}{ccc}
DP_1 & DP_2 & DP_3 \\
\cdot & X & \cdot \\
X & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\end{array}
\]

$DP_1$ and $DP_2$ are inter-dependent, as shown by the two off-diagonal ‘X’s, because they have to be adjusted iteratively to obtain $T_d$ and $Q_d$. An example of $DP_1$ and $DP_2$ is shown in Figure 3.1. The inter-dependent DPs were clustered, which formed two modules, module A and B, as shown in the DSM. This implies that the $DP_1$ designer will have to communicate with the $DP_2$ designer in a team – team A – to handle the inter-dependency. However, they can work independently from the $DP_3$ designer, who is in team B. Therefore, a modular design was achieved, which reduces the complexity of the design task.
3.3 Employing AD prior to DSM to Reduce the Complexity of a Hypothetical Water Faucet System

This subsection explains what the outcome would be, if the system architect employed AD prior to DSM. Below are the FRs (functional requirements) of the water faucet system, which were translated from the customer requirements:

FR1 = Discharge water continuously at desired temperature, $T_d$
FR2 = Discharge water continuously at desired volume flow rate, $Q_d$
FR3 = Accumulate water in sink

The FRs were then translated into DPs (design parameters), and we employed the DPs in the preceding DSM:

DP1 = Hot water knob
DP2 = Cold water knob
DP3 = Drain plug

The design matrix in Equation 3.1 represents the relationships between the FRs and the DPs.
FR1 is affected by both DP1 and DP2, as indicated by the two ‘X’s in row 1. FR2 is also affected by both DP1 and DP2, as indicated by the two ‘X’s in row 2. Hence, the selected DPs cause inter-dependency between FR1 and FR2, which results in a coupled design. Although the DP1 designer can communicate with the DP2 designer to handle the inter-dependency – which is the recommended solution in the preceding DSM – the design tasks will still be highly complex, as iteration is required to fulfill the FRs. Turning the hot water knob to adjust temperature will alter the flow rate, and turning the cold water knob to adjust flow rate will alter the temperature. Moreover, the designers are likely to have to modify FR1 without disrupting FR2 and vice versa. For example, the customers may want to increase $T_d$ but keep $Q_d$ the same. Therefore, removing the functional inter-dependency is optimal to the design process.

To remove the functional inter-dependency, the inappropriate DPs has to be changed. Another set of DPs may be suggested:

DP1 = Water temperature control lever  
DP2 = Water flow rate control lever  
DP3 = Drain plug

Figure 3.2 shows an example of DP1 and DP2. The designers can adjust the water temperature by pivoting the lever in a left-right motion, and adjust the water flow rate by pivoting the lever in an up-down motion. The temperature control will not affect the flow rate, and the flow rate control will not affect the temperature. Hence, every FR is affected by only one DP, as shown in the diagonal design matrix in Equation 3.2.
This is the DSM of the new set of DPs:

\[
\begin{pmatrix}
\text{FR1} \\
\text{FR2} \\
\text{FR3}
\end{pmatrix} =
\begin{pmatrix}
X & 0 & 0 \\
0 & X & 0 \\
0 & 0 & X
\end{pmatrix}
\begin{pmatrix}
\text{DP1} \\
\text{DP2} \\
\text{DP3}
\end{pmatrix}
\] (3.2)

Therefore, an uncoupled design, where FRs are independent of one another, was achieved.

Figure 3.2. Water faucet with temperature control functionally isolated from flow rate control

This is the DSM of the new set of DPs:
Although DP1 and DP2 are functionally independent of each other, they are physically inter-dependent, as indicated by the two off-diagonal ‘X’ s. This is because the water temperature control and the water flow rate control are physically integrated – they share a lever mechanism (Figure 3.2). Hence, the DP1 designer will still have to communicate with the DP2 designer in a team – team A – to decide on a lever mechanism design. Nevertheless, the communication will be much less complex than the one in the initial design (Figure 3.1) where iteration is required. Therefore, although the new DSM appears similar to the initial DSM – and a modular design was achieved once more – the inter-dependency in module A is now weaker.

Furthermore, since the lever mechanism design is unlikely to change, we can constrain it by regarding it as a design rule (DR1), to eliminate the inter-dependency:

```
DR1  DP1  DP2  DP3
DR1      A
DP1  X      B
DP2  X       
DP3      C
```

DP1 and DP2 are now hierarchically dependent on DR1, as indicated by the two ‘X’ s in DR1 column – a lever mechanism has to be selected and installed before using it to adjust the water temperature and flow rate. However, once DR1 is set, the DP1 designer and the DP2 designer can proceed to work independently from each other – the water temperature can be adjusted independently from the water flow rate and vice versa. This is indicated by the absence of off-diagonal ‘X’ s in the DSM, excluding the ‘X’ s in the DR rows and DR columns. Therefore, the information of the lever mechanism design is visible to both designers, but the information of the water temperature adjustment is hidden from the DP2 designer, and the information of the water flow rate adjustment is hidden from the DP1 designer.
3.4 Conclusion of Case Study

The preceding hypothetical case study demonstrates how AD (axiomatic design) and DSM (design structure matrices) complement each other to overcome their shortcomings in reducing system complexity. Although DSM segregates a system into modules that are independent of one another, functional dependencies may be present within the modules, which impede system development, maintenance, and modification. AD can be employed to eliminate these functional dependencies. However, total elimination of functional dependencies in complex systems may be difficult to accomplish. Furthermore, physical dependencies are inevitable. DSM can then be employed to manage these remaining dependencies. Therefore, AD and DSM, when employed together, can effectively reduce system complexity, and thus ease the system development process. Since DESA builds on AD, it complements DSM in the same way.
CHAPTER 4. Revising DESA

4.1 Shortcomings of DESA (Design Equations for Systems Analysis)

Mayer (1997) classified programming knowledge into four levels: syntactic knowledge, semantic knowledge, schematic knowledge, and strategic knowledge. This classification may be applied to knowledge of DESA. Having syntactic knowledge of DESA means knowing what its basic steps are, such as specify FRs and DPs first, and then write design equations. When designers read existing literature of DESA and AD, they can effectively gain syntactic knowledge of DESA, because this knowledge is clearly presented in the literature using case studies. As a result, they would find DESA and its axiom-based theory logical, easy to comprehend, and straightforward to use.

Designers would, however, face many difficulties when trying to use DESA, because its semantic knowledge is nebulously presented in existing DESA and AD literature. Having semantic knowledge of DESA means to profoundly understand the definition of its terms and the rationale of its procedures. When designers read existing literature, they gain this knowledge only at a superficial level – they would know the basic definition of DESA terms and basic rationale of its procedure. Although it is possible to appreciate semantic knowledge of DESA by self analyzing and practicing, this process may be arduous to most designers, and they might not have the time and patience to do it.

A study conducted by Lo and Helander (in preparation), to investigate the usability of DESA, revealed that DESA is difficult to use. During this study, participants were given a short training session in DESA, and then tasked to use it to model three process control configurations. The results revealed inconsistencies in the participants’ specification of functional requirements and design parameters. Three main reasons for the inconsistencies and difficulties in usage were identified. First, there was no clear definition of the terms used. Second, there was no clear description
of the usage procedures. Third, there was no clear explanation of the procedure rationale.

As a result, designers would have to rely much on their intuition when using DESA. This counters the effort of a scientific approach towards engineering design. Furthermore, miscommunication would result, if there is no consensus between designers on the terms and procedure used in a design method. Therefore, this study confirms the need to refine DESA.

Semantic knowledge of DESA has to be clearly presented, so that designers can employ DESA more easily and systematically. To clearly present the semantic knowledge of DESA, the following tasks are necessary:

- Define and distinguish DESA terms to establish a DESA terminology
- Provide straightforward examples to demonstrate the procedure and rationale of DESA

These tasks can be accomplished by analyzing patterns hidden in the usages of DESA, and then present the findings clearly and concisely. It is presumed that design patterns exist and have always existed. It is the goal of scientists and researchers to discover and define these patterns by using systematic empirical methods. The following sections present the results of the tasks.

4.2 Revised DESA Framework

Figure 4.1 shows a revised version of the DESA framework shown in Figure 2.4. There are now three rows of design domains: efficiency, effectiveness, and satisfaction, which are the three elements of usability. The document ISO 9241-11 (1998) Guidance on Usability defines usability as: the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use.
The physical domain was split into two domains: model domain and user interface domain. This is because functional dependency can be caused by the internal structure (model) of a system as well as its user interface. It is essential to locate the source of the dependency to effectively remove it. This is explained in detail in Chapter 6 using a case study. Mapping of design attributes from the model domain to the user interface domain represents user interface design, and mapping from the user interface domain to the action domain represents interaction design.

Figure 4.1. Revised DESA framework
Functional domain and functional specification are now termed as requirement domain and requirement specification, as shown in Figure 4.1. This is because ergonomic requirements and affective requirements were included in the framework, in addition to functional requirements.

4.3 Establishing a DESA Terminology

This section defines and distinguishes key terms in the revised DESA framework (Figure 4.1), which form the DESA terminology.

Usability: the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use

Effectiveness: the functional aspect of products

Functional: useful; well-utilized; primary features

Efficiency: the ergonomic aspect of products

Ergonomic: easy to use; safe to use; physical comfort; mental comfort

Satisfaction: the affective aspect of products

Affective: satisfying, aesthetically pleasing, appealing to emotions

Goal domain: the design domain that contains the identified user goals and user intent; comprise functional user goals, ergonomic user intent, and affective user intent

User goals/intent: what the specified users want to achieve when they use the products; user needs; customer requirements
**Requirement domain:** the design domain that contains the specified product requirements; comprise functional requirements, ergonomic requirements, and affective requirements

**Requirements:** product requirements; what features the product should have to achieve usability

**Model domain:** the design domain that contains the conceptualized model parameters; comprise design parameters (functional parameters), ergonomic parameters, and affective parameters, of the model

**Model parameters:** parameters that constitute the internal structure or underlying workings of the product

**Parameters:** descriptions of the product, especially its structure; constitutes the design plan; each parameter should reduce the number of possible design solutions that can fulfill the corresponding product requirement

**User interface domain:** the design domain that contains the conceptualized user interface parameters; comprise design parameters (functional parameters), ergonomic parameters, and affective parameters, of the user interface

**User interface parameters:** parameters that constitute the user interface of the product

**Action domain:** the design domain that contains the required user actions; comprise functional user actions, ergonomic user actions, and affective user actions

**User actions:** how the specified users have to interact with the product to achieve their goals and intent

**Requirement specification:** mapping of user goals/intent into product requirements. User goals/intent are specified from the point of view of users, whereas product requirements are specified from the point of view of products
Internal structure design: mapping of product requirements into model parameters, i.e. describing the internal structure of the product such that it fulfills the product requirements.

User interface design: mapping of model parameters into user interface parameters, i.e. describing the user interface of the product such that it collaborates with the internal structure to fulfill the product requirements.

Interaction design: mapping of user interface parameters into user actions, i.e. how the specified users should interact with the user interface to achieve their goals/intent.

Besides fulfilling product requirements, parameters also have to satisfy design constraints. It may be difficult to distinguish product requirements from design constraints, as both are necessary attributes of the product. In DESA, product requirements are attributes specified by users and customers, whereas design constraints are attributes arising from limitations (i.e. physical constraints, biological constraints, technological constraints, and resource constraints). Hence, the requirement domain of the DESA framework will only contain product requirements, since all its attributes will be mapped from user goals or user intent (Figure 4.1). Nevertheless, product developers have to be aware of the design constraints, when they map product requirements to product parameters.

Most of the time, users would vaguely specify the product attributes that they desire, which form the user goals and user intent. Product developers will then translate these goals and intent clearly into product requirements. Product parameters that fulfill the product requirements will then be conceptualized. However, users may also specify the exact product parameters that they desire, especially when they are familiar with the product and its structure. Therefore, a set of user interview data can contain a mixture of product requirements and product parameters. Product developers can use the above terminology as a guideline to distinguish the requirements from the parameters.
4.4 Examples of Product Planning and Concept Development using the DESA Framework

This section demonstrates how the DESA framework can be utilized to define and conceptualize usable products. Two examples were presented: water faucet design and microscope workstation design. These examples aid the understanding of DESA’s procedure and rationale.

4.4.1 Usability Design of Water Faucet

User goals/intent:
1. Able to adjust the flow rate of the water – functional
2. Able to adjust the temperature of the water – functional
3. Able to make the adjustments easily – ergonomic
4. Faucet should look impressive and upmarket – affective

Product requirements:
1. Vary water flow rate – functional
2. Vary water temperature – functional
3. Easy variation of water temperature and flow rate – ergonomic
4. Luxurious design – affective

Product parameters, model:
1. Master valve – functional
2. Hot water valve and cold water valve – functional
3. Flow rate valve functionally independent from temperature valve – ergonomic
4. Brass material faucet – affective

Product parameters, user interface:
1. Lever that controls master valve – functional
2. Lever that controls hot water valve and cold water valve – functional
3. Lever action faucet – ergonomic
4. Gold-finish exterior – affective
User actions:
1. Swivel faucet lever left/right to regulate hot and cold water valves – functional
2. Lift faucet lever up/down to regulate master valve – functional
3. Swivel or lift faucet lever to regulate the valves independently – ergonomic
4. No user action required – affective

4.4.2 Usability Design of Microscope Workstation

User goals/intent:
1. Workstation should allow microscope to rest firmly on it – functional
2. Able to rest arm comfortably on the table without abduction – ergonomic
3. Able to maintain a natural back posture – ergonomic
4. Able to rest feet comfortably on the floor – ergonomic
5. Able to move in and out of workstation easily – ergonomic
6. Workstation should occupy little space – affective

Product requirements:
1. Hold microscope firmly – functional
2. Vary elbow-table distance – ergonomic
3. Vary eye-eyepiece distance – ergonomic
4. Vary feet-floor distance – ergonomic
5. Easy entry/exit – ergonomic
6. Compact size – affective

Product parameters, model:
1. Table legs near edges of table top; dense material for table legs – functional
2. Height-adjustable table (pneumatic pump activated) – ergonomic
3. Height-adjustable independent microscope platform (motorized) – ergonomic
4. Height-adjustable footrest (pneumatic pump activated) – ergonomic
5. Chair with castor wheels – ergonomic
6. Extendable table: 60mm length; 80 to 100mm when extended – affective
Product parameters, user interface:
1. No user interface required – functional
2. Pneumatic pump hand-lever beneath table – ergonomic
3. Up/down button to control motorized microscope platform – ergonomic
4. Pneumatic pump foot-lever attached to footrest – ergonomic
5. No user interface required – ergonomic
6. Manual slide-out and lock mechanism – affective

User actions:
1. No user action required
2. Lift pneumatic pump hand-lever to pull/push table up/down – ergonomic
3. Depress up/down button to raise/lower microscope platform – ergonomic
4. Step on pneumatic pump foot-lever to move footrest up/down – ergonomic
5. Push feet against floor to move chair towards table or away from it – ergonomic
6. Slide table extension out and lock it in place – affective
CHAPTER 5. Case Study: Using DESA to Design a More Usable Inspection Method

5.1 Introduction of Case Study

Usability inspection is commonly used in the evaluation of computer system user interfaces. This case study highlights frequently used usability inspection methods. Examples of inspection methods include: heuristic evaluation, guideline review, standards, design rules, and cognitive walkthrough. While understanding the context of use, usability issues of these methods are identified; some methods are profound, some are superficial, and most of them are coupled. Nielsen’s usability heuristics, for example, is a coupled method. Besides being coupled, the heuristics are often profound and poor in diagnosing context specific problems. On the other hand, functional and/or operational inspection methods, such as standards inspection and design rules, identify problems at a superficial level. Hence, this case study suggests the employment of a formal user-centered design method to design inspection methods. It demonstrates how DESA can be used to design inspection methods.

5.2 Overview of Usability Inspection Methods

Over the past two decades, users have become increasingly intolerant of poor usability. Comparing two functional equivalent computer systems, the one with poor user interface (UI) usability will become obsolete, and will be abandoned. Due to this trend, there has been an increasing emphasis on UI usability, which has resulted in the development of numerous usability evaluation methods. These methods were classified by Ivory and Hearst (2001) into five categories: testing, inspection, inquiry, analytical modeling, and simulation.

Among these five classes of methods, inspection is the only one that does not involve users. Analytical modeling and simulation may sometimes exclude users. However, to create accurate user models, evaluators must have knowledge of users. Hence, users
are indirectly involved in these methods. It is good to have users participate in usability evaluation and feedback from them is valuable. Nevertheless, in situations where users cannot be involved, evaluators will have to rely on inspection methods to evaluate the UI.

A large number of inspection methods are available. On the surface, most of them appear quite similar to one another. Upon closer examination they can actually be segregated into several distinctive groups (Nielsen, 1994; Vanderdonckt, 1999; Ivory and Hearst, 2001):

- Heuristic evaluation
- Guideline review
- Style guides
- Standards inspection
- Design rules
- Cognitive walkthrough
- Pluralistic walkthrough

In heuristic evaluation, a small group of usability experts examines the GUI and assess it based on a set of usability heuristics. Nielsen's 10 heuristics is an example of a set of usability heuristics (Nielsen, 1994). Guidelines typically have a much longer list of items than heuristics, and it serve as a form of usability checklist. It is useful for less experienced GUI evaluators. Both heuristics and guidelines are generic and thus are applicable to a wide-range of GUIs. In contrast, style guides are specific guidelines developed for usage on distinct GUIs with the aim of design consistency. Standards are official specifications published by organizations for standardization with the aim of standardizing design. An example of usability standards is the ISO 9241-11. Design rules are straightforward specifications written in a form that requires no further interpretation. Hence, each set of design rules can only be used for the specific GUIs that they are written for. In cognitive walkthrough, evaluators identify design flaws by thinking about the probable mental processes of users when they interact with the GUI. Pluralistic walkthrough is multiple evaluators conducting cognitive walkthrough.
Unfortunately, inspection methods may themselves suffer from usability problems, as addressed below. The objective of this case study is, therefore, to demonstrate how DESA can be employed to design a more usable inspection method.

5.3 Usability Problems of Inspection Methods

Before proceeding to design a more usable inspection method, it is necessary to understand usability problems with existing inspection methods. For this purpose, related literature is first reviewed.

Several studies have compared usability evaluation methods (Desurvire et al. 1991, Jeffries et al. 1991, Karat et al. 1992 and Virzi et al. 1993). These studies suggest that heuristic evaluation and user testing are the most effective methods for identifying usability problems. Hence, they are frequently employed by evaluators. On the other hand, when concluding which of the two is the more effective method, the studies gave contradictory recommendations. This can be explained by the results of an experiment conducted by Fu et al. (2002). The aim of this experiment was to study the difference between heuristic evaluation experts and user testing subjects in identifying usability problems of software interface. Results of this experiment revealed that heuristic evaluation experts identify significantly more low performance level problems (skill-based and rule-based problems), while user testing subjects identify significantly more high performance level problems (knowledge-based problems). Hence, the previous studies demonstrate contradictory results because they utilized tasks of different levels of difficulty — studies that utilized skill-based and rule-based task revealed that heuristic evaluation is more effective than user testing, while studies that utilized knowledge-based task revealed that user testing is more effective than heuristic evaluation.

Several usability issues concerning inspection methods have been reported. Some methods are, as we mentioned, coupled. For example, Nielsen’s usability heuristics is a coupled method; see Figure 5.1 (Helander, 2003). Ideally there should be a one-to-
one correspondence between the ergonomic requirements (ERs) and ergonomic parameters (EPs) in Figure 5.1. However, most of the ERs were satisfied by several EPs. This makes a coupled process; we may find that one value of an EP will satisfy one ER but not another ER. By reengineering the ERs (Lo and Helander, 2004a), a less coupled matrix was generated (Figure 5.2). The number of couplings was reduced from 22 Xs in the matrix to 17 Xs. In addition, only the inner square of the matrix was coupled, which reduced the search for solutions.

| ER, : Keep users informed about system status | EP, : Provide feedback |
| ER, : Match between system and the real world | EP, : Speak user's language |
| ER, : User control and freedom | EP, : Provide emergency exit |
| ER, : Consistency | EP, : Follow platform conventions |
| ER, : Facilitate recognition rather than recall | EP, : Visibility of objects, actions, and options |
| ER, : Flexibility and efficiency of use | EP, : Facilitate accelerators |
| ER, : Help users recognize, diagnose, and recover from errors | EP, : Suggest solutions in error messages |

Figure 5.1. Coupled Nielsen's usability heuristics

Constraint C, = Minimalist design

| ER, : Understandability of vocabulary | EP, : Conventional or user familiar words |
| ER, : Guidance for error recovery | EP, : Suggest solutions in error messages |
| ER, : Consistency of behavior | EP, : Follow platform conventional behavior |
| ER, : System transparency | EP, : Visibility of objects, actions, options, and system states |
| ER, : Match between system behavior and the real world | EP, : User familiar concepts and techniques |
| ER, : User control and freedom | EP, : Provide emergency exits |
| ER, : Flexibility and efficiency of use | EP, : Facilitate accelerators |

Figure 5.2. Less coupled Nielsen's usability heuristics. Only the small matrix in the center is coupled

Mahemoff and Johnston (1998), and Dix et al. (1998) reviewed guidelines for usability assessment and found several usability problems in using the guidelines; Vanderdonckt (1999) enumerated 27 problems after going through five development milestones of guidelines. Generally, the problems are:

- Guidelines are often either too profound or too superficial
- Difficult to select appropriate guidelines
• Difficult to apply guidelines to specific context
• Difficult to interpret the intended prescription of guidelines
• Conflict between guidelines, and validity of guidelines

Souza and Bevan (1990) conducted an empirical test to evaluate a set of usability standards drafted by ISO (International Standards Organisation). Designers were asked to study the standards, and use them to redesign a menu interface. The designers made errors or had difficulties with 91% of the standards. Thovtrup and Nielsen (1991) conducted two empirical tests. In a laboratory experiment students were asked to use a two page interface standard to design an interface. Then there was a field experiment which studied developers’ usage of their company’s interface standard. The students achieved only 71% compliance with the two page standard. The developers, while using their company’s standards, were able to find only 4 of 12 deviations from the standards. Three of their software products violated 7 to 12 of the 22 mandatory rules in the standard.

This review suggests that user testing and heuristic evaluation are the most usable UI (user interface) evaluation methods. Hence, if evaluators are to choose between several evaluation methods, it is likely that they will select user testing or heuristic evaluation. This also implies that if evaluators decide to employ inspection methods and have to select one, it is likely that they will select heuristic evaluation.

5.4 Designing a More Usable Inspection Method

As stated in the literature review, user interface evaluators encounter many usability issues in using inspection methods. To improve the inspection methods, user-centered design methods must be employed to redesign them. During the redesign process, the inspection methods are then treated as products and the UI evaluators are the users.

In the following sections, we describe a procedure for designing a usable inspection method. We first interviewed UI evaluators to understand the usage context of
inspection methods. In the second stage, we employed DESA to specify the method’s requirements and parameters.

5.4.1 Interviewing UI (user interface) Evaluators to Understand the Context of Use of Inspection Methods

From a review of literature, several usability issues of inspection methods were identified. However, since we did not understand how these methods were used, two UI evaluators were interviewed so that we could document the context of usage of inspection methods. One of the UI evaluators was a human factors consultant with much experience in evaluating clients’ UIs. The other was a human factors engineer who would frequently evaluate the UIs that he had designed. The interviews were conducted face to face and through email. Examples of questions asked are:

- What are the common procedures employed by UI evaluators when they evaluate UI?
- Are inspection methods usually used and in what context are they used?
- Is heuristic evaluation useful and easy to use, and in what context is it used?
- Are automatic capture and analysis tools, base on inspection methods, frequently used?
- You have mentioned that you mix and match guidelines from different sources. May I know from which sources do you usually get your guidelines?
- After evaluating UIs using guidelines, do you use other evaluation methods to evaluate further?

The interviews revealed that heuristic evaluation and user testing are the most frequently used evaluation methods. This finding is in agreement with previous studies. The interview also revealed a common procedure for usability evaluation. The inspection method is employed followed by user testing. The preference for this procedure is evident if we examine Fu’s (2002) findings – inspection methods were more effective in identifying low performance level problems (skill-based and rule-based problems), while user testing was more effective in identifying high performance level problems (knowledge-based problems. Therefore, many evaluators
use inspection methods to eliminate low performance level problems, so as to focus on high performance level problems during user testing.

Based on Fu’s (2002) findings, user testing alone would be inadequate, as it cannot identify many low performance level problems; it should be used together with inspection methods to identify the maximum number of problems. It may, however, be erroneous to judge an evaluation method’s effectiveness based on the number of problems identified. The validity of the identified problems should hold more weight than the quantity. Based on our interview results, the latest trend in the UI (user interface) design field is: if users do not report predicted problems, then they are not usability problems – many low performance level problems identified by inspection methods are, thus, invalid, and user testing alone is the final measure. As a result, some evaluators with time and/or budget constraints proceed straightaway to user testing.

One might question whether inspection methods can also be conducted alone. This procedure is usually employed when circumstances do not permit users’ involvement. Usually, however, inspection methods are used together with other methods. This is because inspection methods cannot identify many high performance level problems that are mostly valid, and cannot assess usefulness of UIs. Then again, one might question whether it is possible to develop an inspection method that can identify high performance level problems, and assess both ease of use and usefulness of UI. To answer this question, further research and investigation are needed.

The interview also revealed that although the evaluators had numerous inspection methods to choose from, only heuristic evaluation and guideline review are frequently used. This implies that heuristic evaluation and guideline review are perceived of as more useful compared to other inspection methods. Nielsen (1995) showed that the usage frequency of inspection methods correlate positively with its perceived usefulness. Nielsen’s findings also showed that heuristic evaluation and user testing are the most frequently used evaluation methods, and thus most useful, which again coincides with the interview results and previous studies. Nielsen’s study states six requirements for a frequently used inspection method:
1. Provides information that is useful in improving UI
2. Inexpensive
3. Fast to use
4. Easy to learn
5. Flexible and adapt to specific context
6. Aggressive advocacy

Requirement 1 and 5 correspond to effectiveness or usefulness of the inspection method, whereas requirement 3 and 4 correspond to efficiency or ease of use of the inspection method. Hence, most of these requirements correspond with inspection methods' usability issues identified in the literature review.

Sometimes, UI evaluators employ several inspection methods together. Some evaluators may select a complete set of heuristics or guidelines, while others prefer to compare several sets, and select relevant prescriptions from each set.

The interview results in the preceding paragraphs show that inspection methods – particularly heuristic evaluation and guideline review – are frequently employed for usability evaluation of UI. They also show that unless users' involvement is impractical, inspection methods are usually used together with user-participatory methods, and user testing is the most frequently employed method in such cases.

5.4.2 Employing DESA to Design the Inspection Method

With supporting information from the literature review and the interview, we proceeded to employ DESA to the design process, which can be summarized into the following five steps:

1. Identify user goals and user intent
2. Specify product requirements
3. Conceptualize product parameters
4. Specify required user actions
5. Decompose the product requirements and the product parameters into a lower level

The results corresponding to first four steps are summarized in Table 5.1. The product parameters were not separated into internal structure (model) parameters and user interface parameters, because it is unnecessary to separate an inspection method into internal structure and user interface. This is unlike a software system – or other systems – which can be separated into model subsystem and user interface subsystem. Such separation is explained in detail in Chapter 6, using a case study.

Table 5.1. Designing an inspection method using DESA

<table>
<thead>
<tr>
<th>User goals/intent</th>
<th>Product requirements</th>
<th>Product parameters</th>
<th>User actions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functional</strong></td>
<td>Able to evaluate usability of UI (user interfaces) effectively</td>
<td>Effectively evaluate usability of UI</td>
<td>UI design patterns: a design pattern describes the best common solution to a common problem within a specific context</td>
</tr>
<tr>
<td><strong>Ergonomic</strong></td>
<td>Able to employ the method easily</td>
<td>Easy usage</td>
<td>Clearly and concisely written design patterns</td>
</tr>
</tbody>
</table>

The functional requirements and corresponding design parameters, listed in Table 5.1, were decomposed into a lower level, as shown in Figure 5.3. The ergonomic requirement and corresponding ergonomic parameter were also decomposed into a lower level, as shown in Figure 5.4.
DESA: A Design Method that Reduces System Complexity

Effectively evaluate usability of UI

Evaluate usefulness of UI
Evaluate ease of use of UI
Evaluate satisfaction of UI
Contextually relevant evaluation
Provide constructive recommendations

UI design patterns

Write design patterns for usefulness of UI
Write design patterns for ease of use of UI
Write design patterns for satisfaction of UI
Write design patterns for various usage contexts
Write design patterns for high abstraction level problems

Figure 5.3. Decomposing the functional requirement and the design parameter

Easy usage

Easy to learn
Fast to use
Intuitive prescriptions

Clearly and concisely written design patterns

Step-by-step UI design prescription
Consistent design pattern presentation format; "Search" function
Design patterns include problems encountered, usage contexts, prescribed solution, solution rationale, and a pictorial example

Figure 5.4. Decomposing the ergonomic requirement and the ergonomic parameter
5.5 Discussion of Case Study

Based on the above results, there are eight requirements for a usable inspection method (Table 5.2). Designers of inspection methods can use this set of requirements to systematically evaluate existing inspection methods, and use the results to redesign the methods.

Table 5.2. Eight requirements for a usable inspection method

<table>
<thead>
<tr>
<th>No.</th>
<th>Requirements</th>
<th>Requirement type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Evaluate usefulness of UI</td>
<td>Functional</td>
</tr>
<tr>
<td>2.</td>
<td>Evaluate ease of use of UI</td>
<td>Functional</td>
</tr>
<tr>
<td>3.</td>
<td>Evaluate satisfaction of UI</td>
<td>Functional</td>
</tr>
<tr>
<td>4.</td>
<td>Contextually relevant evaluation</td>
<td>Functional</td>
</tr>
<tr>
<td>5.</td>
<td>Provide constructive recommendations</td>
<td>Functional</td>
</tr>
<tr>
<td>6.</td>
<td>Easy to learn</td>
<td>Ergonomic</td>
</tr>
<tr>
<td>7.</td>
<td>Fast to use</td>
<td>Ergonomic</td>
</tr>
<tr>
<td>8.</td>
<td>Intuitive prescriptions</td>
<td>Ergonomic</td>
</tr>
</tbody>
</table>

UI (user interface) evaluators can also use the eight requirements to compare existing evaluation methods, and rationally decide the most appropriate method for the evaluation context. This can be done by inputting the eight requirements and the parameters of each method into the QFD’s (Quality Function Deployment) “house of quality” (Hauser and Clausing, 1988). After rating the importance of each requirement, the evaluators can then make rational comparisons between the different methods. It also clearly displays trade-offs between the parameters of different methods.
UI design patterns, suggested in the above design of an inspection method, may not be the best solution. There may be other solutions that better fulfill the eight requirements. Nonetheless, design patterns are increasingly employed to design and evaluate UI. Yahoo! has designed and built a pattern library that contains web UI design patterns: http://developer.yahoo.com/ypatterns.

Besides inspection methods, DESA may be used to design other types of usability evaluation methods such as user testing, inquiry, analytical modeling, and simulation. The procedure may be similar to the one employed in this study, and the evaluation methods' requirements may coincide with many of the above eight requirements.

DESA is particularly effective in designing for usability, as demonstrated in this case study. This is because DESA has a user-centered framework, which can be used to design for effectiveness, efficiency, and satisfaction of product, which are the three elements of usability. DESA achieves these three elements by ensuring that product parameters fulfill functional requirements and ergonomic requirements. Furthermore, DESA reduces complexity by minimizing dependencies between product requirements – a simple design is more usable than a complex design. This is demonstrated in detail via a case study in Chapter 6. Therefore, DESA can be employed in the product development process when usability is a concern.

The revised DESA framework and established DESA terminology, in Section 4.2 and Section 4.3 respectively, resulted in a systematic employment of DESA – requirement specification and solution conceptualization became easier, less ambiguous, and more consistent. The proposed terminology may be applicable to other design methods that aid requirement specification and solution conceptualization. For example, in QFD, customer requirements have to be specified, and design parameters have to be suggested. However, it may be difficult to distinguish design parameters from customer requirements. Moreover, customer requirements may contain design constraints. The proposed terminology can be used as a guideline to distinguish the terms, which will solve these problems.
5.6 Conclusion of Case Study

This case study reviewed usability issues in the use of inspection methods. The author advocates the use of user-centered design methods to design inspection methods that in themselves have good usability qualities. The employment of DESA is supported with information from the review of existing inspection methods and an interview of UI (user interface) designers. Inspection method designers can consider employing the procedure proposed in this case study. Therefore, besides being employed to design human-machine systems (Section 4.4; Helander, 2007; Lo and Helander, 2007), DESA can also be employed to design tools and methods, as demonstrated in this case study.
CHAPTER 6. Case Study: Using DESA to Reduce the Complexity of Object-Oriented Software

6.1 Introduction of Case Study

To reduce complexity in software systems it is essential to minimize the functional dependencies in them. Functional dependency can be caused by the internal logic (model) of the system as well as the user interface. It is then vital to locate the source of the dependency, so that it can be removed. DESA (Design Equations for Systems Analysis) offers an opportunity to accomplish this. It allows separate examination of the model and the user interface when evaluating functional dependencies. This study investigates this potential of DESA in identifying coupled relationships. We used an object-oriented game application as a case study. DESA was found to effectively reduce the complexity of object-oriented software systems.

6.2 Dependencies in Software Systems

A software system is inherently complex due to many dependencies between the various components that constitute the software. The dependencies between components impede maintenance, modification, and extension, which are constantly required in software systems. To minimize these dependencies, object-oriented design, which is a prevalent software design method, can be employed (Wirfs-Brock et al., 1990). C++ and Java are programming languages that conform to this method, and have the potential to increase the modularity of software systems. Modularity is defined as a particular design structure, in which parameters and tasks are interdependent within units (modules) and independent across them (Baldwin and Clark, 2000).
6.3 Object-Oriented Software Design and its Shortcoming

Any software system has a set of responsibilities to fulfill. In object-oriented design, these responsibilities are distributed among classes, which are further distributed among methods within each class. Every method is composed of two types of software code: specification and implementation. The former defines the responsibilities, while the latter implements the responsibilities.

During run-time, one or more instances of every class are created, and these instances are termed objects. Hence, a class can be perceived as a blueprint of one or more objects, and objects from the same class have the same set of responsibilities. Classes and their objects are often modeled after real-world entities. For example, for a game such as chess, the class Player may be employed to model players, and the class Piece may be employed to model chess pieces. Piece may be assigned the following responsibilities: know this Piece’s color, know this Piece’s rank, and change this Piece’s location on the chessboard. Thirty-two Piece objects will be instantiated to represent the thirty-two chess pieces.

To support the system functionality, objects within a software system have to collaborate. In order to execute a method within an object, it has to be invoked by another object, and data may be transferred in the process. If object A invokes a method of object B, object A is termed as client, while object B is termed as server. Object B can also assume the role of a client to a few other objects, including object A. Object-oriented design keeps software complexity under control by minimizing the client-server dependency. Classes are written such that a client depends on the specification, and not implementation of the server. A client knows what methods the server has, not how the methods are executed. In this way, modifying and extending methods of a server will cause minimal disruption to its clients.

Another technique to minimize client-server dependency is to ensure unidirectional dependency between two objects. If object A is designed to be a client of object B, we
should avoid making object B also a client of object A. If not, inter-dependency will occur, which increases software complexity.

However powerful in increasing software modularity, object-oriented design alone will not reduce software complexity (Suh, 2001; Oktay, 2006). This is because in object-oriented design, software concerns are intuitively separated into distinct entities - often based on experience. However, to reduce complexity software concerns must also be explicitly separated into functions. A minimally complex software system will allow functions to be modified or added independently, without disrupting other functions. This will then ease maintenance, modification, and extension (Suh, 2001). Such functional independence is not ensured in object-oriented design - for example, one class may contain two or more functions, which are highly inter-dependent, as dependencies are not controlled within a class. Therefore, object-oriented design may still produce complex software systems. In other words, object-orientation offers the necessary but not the sufficient conditions for reducing complexity. A software design method that overcomes this shortcoming is presented in the following subsection.

6.4 Complementing Object-Oriented Design with Axiomatic Design

ADOSS (Axiomatic Design of Object-oriented Software Systems) is a software design method that minimizes dependencies between functions of an object-oriented software system (Do and Suh, 2000b). It utilizes axiomatic design which to minimize dependencies between functions of a complex system (Suh, 1990). ADOSS is summarized in the following paragraphs.

ADOSS employs a V model for software design (Figure 6.1) (El-Haik, 1999). The left side of the model represents a top-down approach in building the software hierarchy, in which axiomatic design is employed; the right side represents a bottom-up approach in building the object-oriented model, in which object-oriented design is employed. The V model comprises the following detailed steps:
1. Define FRs (functional requirements) of the software system. Identify customer needs of the system, and map them into FRs. Each FR can represent an object.

2. Mapping between domains and the independence of software functions. Map every FR into a DP (design parameter). DPs are design solutions in the form of data or input for objects.

3. Decomposition of FRs and DPs. FRs are decomposed, and the results are mapped into DPs again. This decomposition process is repeated until all DPs are explicit enough to be implemented. The resultant decomposition hierarchies of FRs and DPs represent the software system architecture.

4. Definition of modules / complete design matrix. A design matrix is constructed to provide a gestalt representation of the relationship between the FRs and the DPs (Figure 6.2). Each row of the matrix constitutes a module of the software system. Hence, a module explicitly represents an FR – unlike object-oriented design where a module represents a class or a real entity. As a result, high modularity implies minimal dependencies between FRs, which implies low complexity.
5. Identify objects, attributes, and operations. The FR-DP pairs in the completed design matrix are translated into object-oriented design classes which comprise data and methods.

<table>
<thead>
<tr>
<th>FR1 Manage design workflow</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>DP1 Design roadmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR2 Provide decision-making environment</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>DP2 Provide decision-making criterion</td>
</tr>
<tr>
<td>FR3 Provide efficient data I/O</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>DP3 Data manager</td>
</tr>
<tr>
<td>FR4 Provide utility function</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>DP4 Plug-in software</td>
</tr>
<tr>
<td>FR5 Support ease of use while using software</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>DP5 Graphical user interface (GUI)</td>
</tr>
</tbody>
</table>

Figure 6.2. Design matrix of the Acclaro software (Do and Suh, 2000a)

The ADOSS software design method was employed in the development of a commercial software system called Acclaro (Do and Suh, 2000a). Acclaro is an interactive and general-purpose software package for designers who practise axiomatic design.

6.5 Shortcoming of ADOSS (Axiomatic Design of Object-oriented Software Systems)

An object-oriented software system can be decomposed into two subsystems: model and user interface (Nino and Hosch, 2005). The model subsystem comprises objects that are responsible for the internal logic of the system. The user interface subsystem comprises objects that are responsible for displaying model state to the user, and for getting user input to the model.

In ADOSS, both the model subsystem and the user interface subsystem are denoted as DPs (design parameters), which are intended to fulfill various FRs (functional requirements). For example, the first 4 DPs in Figure 6.2 denote the model, while DP5 denotes the user interface. Hence, when constructing the design matrix to evaluate functional dependencies, the model is examined jointly with the user interface. This particular procedure is inappropriate for three reasons.
First, an FR of a software system is often fulfilled by both the model and the user interface, in collaboration. For example, if an FR is to allow a user to configure image size, the user interface will be responsible for enabling user to input the size, and the model will be responsible for getting the user input from the user interface and know the input value. Therefore, each FR should have two semantically different DPs – one to denote the model, and the other to denote the user interface.

Second, since an FR is often fulfilled by both the model and the user interface, a dependency between two FRs can be caused by either the model or the user interface, or both. It is essential to identify the source of this dependency to effectively remove it. Therefore, the model and the user interface should be examined separately when evaluating functional dependencies.

Third, the model is independent of the user interface, but the user interface is dependent on the model. This unidirectional dependency is inevitable, and it results in the situation that all user interface DPs are dependent on all model DPs. Therefore, if the model is examined jointly with the user interface, the design matrix will be cluttered with many inconsequential ‘X’s, as shown in the last row of the design matrix in Figure 6.2. By examining the model and the user interface separately, these inconsequential ‘X’s will be removed (Figure 6.3).

<table>
<thead>
<tr>
<th>FR1 Manage design workflow</th>
<th>X</th>
<th>0</th>
<th>0</th>
<th>DP1 Design roadmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR2 Provide decision-making environment</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>DP2 Provide decision-making criterion</td>
</tr>
<tr>
<td>FR3 Provide efficient data I/O</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>DP3 Data manager</td>
</tr>
<tr>
<td>FR4 Provide utility function</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>DP4 Plug-in software</td>
</tr>
</tbody>
</table>

Figure 6.3. Design matrix of the Acclaro software, excluding the user interface
6.6 Complementing Object-Oriented Design with DESA

DESA (Design Equations for Systems Analysis) is a design method, which has been demonstrated to be effective in minimizing functional dependencies within human-machine systems, by examining both the internal structure and the user interaction of the systems (Helander, 2007; Lo and Helander, 2007). Since DESA builds on axiomatic design, it can complement object-oriented design in an approach similar to ADOSS (Axiomatic Design of Object-oriented Software Systems). However, there are two fundamental differences between DESA and ADOSS.

First, DESA utilizes a user-centered design model (Figure 6.4), where user goals (UGs) are mapped into FRs (functional requirements), followed by DPs (design parameters), and finally into user actions (UAs) – we only focus on the effective or functional aspect of DESA in this case study. This is different from the ADOSS’ V-model (Figure 6.1), where customer attributes are mapped into FRs, and finally into DPs.

Second, DESA has two DP domains: model domain and user interface domain. This allows separate examination of the model and the user interface when evaluating functional dependencies. In contrast, ADOSS has only one DP domain that contains
both the model DPs and the user interface DPs. Hence, DESA has the potential to
overcome the shortcoming of ADOSS presented in the preceding subsection. This
potential was further investigated via a case study, in which DESA was employed to
evaluate functional dependencies within an object-oriented application termed as Nim
Game. The case study is presented in the following sections.

6.7 Nim Game Specification using DESA Design Model

The Nim Game was obtained from a textbook that introduces object-oriented software
design in Java programming language (Nino and Hosch, 2005). In this application, a
user player takes turn with a computer player to remove sticks from a pile of sticks,
via a graphical user interface. The player who removes the last stick loses. The user is
able to configure the game by specifying the number of sticks to begin with and
which player plays first. The application will display the number of sticks left in the
pile, display the number of sticks last taken by each player, and report the winner
when the game is over.

Figure 6.5 shows the screenshots of Nim Game. The “Configure Game” dialog will be
displayed when the user selects “New Game” in the “Game” menu. The “Game Over”
option pane will be displayed after the last stick is removed.

![Figure 6.5. Screenshots of Nim Game application](image)
The DESA design model, as shown in Figure 6.4, was employed to aid mapping of Nim Game's user goals to its functionality, to its model specification, to its user interface specification, and to its user actions.

The first-level UGs (user goals), in the UG decomposition hierarchy, were:

UG1 = Configure game
UG2 = Take turns with computer to remove sticks
UG3 = View game state

These UGs were then mapped into FRs (functional requirements) of the application:

FR1 = Allow configuration of game
FR2 = Take turns with user to remove sticks
FR3 = Display game state

The difference between the UGs and the FRs is in the point of view – the UGs were explicitly specified from the point of view of user, while the FRs were explicitly specified from the point of view of the application.

The FRs were mapped into DPₘs (model design parameters) of the application:

DPₘ₁ = Game
DPₘ₂ = Player -- Pile::remove()::sticks to take
DPₘ₃ = Game -- Game

Each fully specified DPₘ contains three types of information: the class responsible for fulfilling the functional requirement, the class method that implements the responsibility, and the data of concern. For example, in DPₘ₂, Pile is the class responsible for fulfilling FR2, remove is Pile's method that removes sticks, and sticks
to take is the data of concern. Besides concrete classes, the DP\_m\_s may also comprise abstract classes or interfaces.

Each DP\_m can have a few responsibilities, and they were separated using the symbol "\_\_". For example, in DP\_m\_2, Player is the interface responsible for determining the number of sticks to remove and then command a Pile object to remove them, while Pile is the class responsible for removing these sticks. However, Player's method was not specified at this stage because the types of players and their strategies were yet unknown. This implies that FR\_2 had to be decomposed into second-level FR\_s. Due to similar reasons, FR\_1 and FR\_3 were also decomposed.

The responsibilities of each DP\_m were listed for documentation purpose, and they were specified in \(<\text{type of object}>:<\text{responsibility}>\) format:

- DP\_m\_1 = Game: get initialization data from controller
- DP\_m\_2 = Interface player: determine number of sticks to take and command pile to remove sticks
  - Pile: remove sticks
- DP\_m\_3 = Game: notify observers when game changes state
  - Game: know game state information

The DP\_m\_s were mapped into DP\_ui\_s (user interface design parameters) of the application:

- DP\_ui\_1 = ConfigurationDialog::ConfigurationDialog() -- ConfigurationPanel::okPanel() -- Anonymous::actionPerformed(), NimController
- DP\_ui\_2 = NimInterface -- NimInterface -- NimController::sticks to take
- DP\_ui\_3 = NimInterface -- NimInterface -- NimInterface

The DP\_ui\_s and the DP\_m\_s have similar specification syntax. However, they are different from a semantic perspective. While the DP\_m\_s are responsible for implementing the internal logic of the application, the DP\_ui\_s are responsible for implementing the user interface.
The responsibilities of each DP$_{ui}$ were also listed for documentation purpose:

DP$_{ui1}$ = View: display dialog for user to input initialization data  
Controller: listen to "OK" button of dialog for initialization data  
Controller: pass initialization data to game

DP$_{ui2}$ = View: display components for interface player to remove sticks  
Controller: listen to components for number of sticks to remove  
Controller: pass number of sticks to remove to interface player

DP$_{ui3}$ = View: display components for user to view game state  
View: observe game to update state changes  
View: query game state and write it to components, when game changes state

The DP$_{uis}$ were then mapped into UAs (user actions), which are actions that a user has to perform to achieve the UGs:

UA$_1$ = Interact with "Configure Game" dialog that pops up after clicking on "New Game" menu item of "Game" menu. Click on "OK" button after configuration

UA$_2$ = When text field of "Computer takes" panel is highlighted, wait for computer to remove sticks. When text field of "User takes" panel is highlighted, interact with remove stick panel

UA$_3$ = View panels that display game state

The first-level user goals and functional requirements were decomposed, and the mapping process was repeated. The decomposition ended at second-level, because the DP$_m$s, DP$_{ui}$s, and UAs had been fully and clearly specified. Table 6.1 shows the first and second-level UGs, FRs, DP$_m$s, DP$_{ui}$s, and UAs. Responsibilities of each DP$_m$ and each DP$_{ui}$ are displayed in Table 6.2.
Table 6.1. Decomposed UGs, FRs, DPₘₛ, DPₜₛ, and UAs

<table>
<thead>
<tr>
<th>UGs</th>
<th>FRs</th>
<th>DPₘₛ</th>
<th>DPₜₛ</th>
<th>UAs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Configure game</td>
<td>Allow configuration of game</td>
<td>Game</td>
<td>ConfigurationDialog::ConfigurationDialog()</td>
<td>Interact with &quot;Configure Game&quot; dialog</td>
</tr>
<tr>
<td></td>
<td>Allow specification of number of sticks</td>
<td>Game::Game() -- Pile::Pile()::starting sticks</td>
<td>ConfigurationPanel::PileSticks()</td>
<td>that pops up after clicking on &quot;New Game&quot; menu item of &quot;Game&quot; menu. Click on &quot;OK&quot; button after configuration</td>
</tr>
<tr>
<td></td>
<td>to start</td>
<td></td>
<td>Anonymous::actionPerformed(), NimController::setStartingSticks(), initializeGame()::starting sticks</td>
<td>Enter number of sticks to start in text field of &quot;Number of sticks to start with&quot; panel</td>
</tr>
<tr>
<td>1.1 Specify number of sticks to start</td>
<td>Allow selection of user plays first or not</td>
<td>Game::Game::first player</td>
<td>ConfigurationPanel::firstPlayerPanel() --</td>
<td>Click on &quot;User plays first&quot; radio button or &quot;Computer plays first&quot; radio button</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NimController::setUserPlaysFirst(), initializeGame()::first player</td>
<td></td>
</tr>
<tr>
<td>2 Take turns with computer to remove sticks</td>
<td>Take turns with user to remove sticks</td>
<td>Player::remove():stick to take</td>
<td>NimInterface::NimInterface::sticks to take</td>
<td>When text field of &quot;Computer takes&quot; panel is highlighted, wait for computer to remove sticks. When text field of &quot;User takes&quot; panel is highlighted, interact with remove stick panel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NimController::setNumberToTake(), takeTurn()::remove():sticks to take</td>
<td>Automated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NimController::setNumberToTake(), takeTurn()::remove():sticks to take</td>
<td>Click on &quot;Take 1&quot; button, &quot;Take 2&quot; button, or &quot;Take 3&quot; button</td>
</tr>
<tr>
<td>3 View game state</td>
<td>Display game state</td>
<td>Game</td>
<td>NimInterface::NimInterface::sticks to take</td>
<td>View panels that display game state</td>
</tr>
<tr>
<td></td>
<td>Display number of sticks last taken</td>
<td>Game::play() -- game::sticksTaken(), Pile::Pile()::sticks last taken</td>
<td>PlayerPanel::PlayerPanel() -- PlayerView::PlayerView() -- PlayerView::update():sticks last taken</td>
<td>View text field of &quot;Computer takes&quot; panel and text field of &quot;User takes&quot; panel</td>
</tr>
<tr>
<td>3.1 View number of sticks last taken</td>
<td>Display number of sticks left in pile</td>
<td>Game::play() -- Game::stickLeft(), Pile::Pile()::sticks left in pile</td>
<td>ReportPanel::ReportPanel() -- NimController::initializeGame() -- NimInterface::update():sticks left in pile</td>
<td>View &quot;Sticks left in pile&quot; panel</td>
</tr>
<tr>
<td></td>
<td>Display number of sticks left in pile</td>
<td>Game::play() -- Game::sticksLeft(), Pile::Pile()::sticks left in pile</td>
<td>NimController::initializeGame() -- NimInterface::update():sticks left in pile</td>
<td></td>
</tr>
<tr>
<td>3.2 View number of sticks left in pile</td>
<td>Report winner of game</td>
<td>Game::gameOver() -- Game::winner()::player who won</td>
<td>NimController::initializeGame() -- NimInterface::reportWinner():player who won</td>
<td>View message of &quot;Game Over&quot; option pane that pops up when game is over</td>
</tr>
</tbody>
</table>
### Table 6.2. Responsibilities of the design parameters

<table>
<thead>
<tr>
<th>DP&lt;sub&gt;ps&lt;/sub&gt; (Design Parameters, Model)</th>
<th>DP&lt;sub&gt;ps&lt;/sub&gt; (Design Parameters, UI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Game: get initialization data from controller</td>
<td>View: display dialog for user to input initialization data</td>
</tr>
<tr>
<td>1.1 Game: get starting sticks from controller and pass them to pile Pile: get starting sticks from game</td>
<td>Controller: listen to &quot;OK&quot; button of dialog for initialization data</td>
</tr>
<tr>
<td>1.2 Game: get user plays first or not from controller</td>
<td>Controller: pass initialization data to game</td>
</tr>
<tr>
<td>2 Interface player: determine number of sticks to take and command pile to remove them Pile: remove sticks</td>
<td>View: display text field for user to specify starting sticks</td>
</tr>
<tr>
<td>2.1 Computer player: command pile to remove 1 stick in every turn Pile: remove sticks</td>
<td>Controller: know starting sticks specified in text field</td>
</tr>
<tr>
<td>2.2 User player: get number of sticks to take from controller and command pile to remove them Pile: remove sticks</td>
<td>Controller: pass starting sticks to game</td>
</tr>
<tr>
<td>3 Game: notify observers when game changes state Game: know game state information</td>
<td>View: display radio buttons for user to select user plays first or not</td>
</tr>
<tr>
<td>3.1 Interface player: notify observers when sticks are removed Abstract class player: get number of sticks last taken from concrete players and know this number</td>
<td>Controller: listen to radio buttons for user plays first or not</td>
</tr>
<tr>
<td>3.2 Game: notify observers when sticks left in pile changes Game, pile: know sticks left in pile</td>
<td>Controller: pass user plays first or not to game</td>
</tr>
<tr>
<td>3.3 Game: notify observer when play is made Game: know game is over or not Game: know winner of game</td>
<td></td>
</tr>
<tr>
<td></td>
<td>View: display components for interface player to remove sticks</td>
</tr>
<tr>
<td></td>
<td>Controller: listen to components for number of sticks to remove</td>
</tr>
<tr>
<td></td>
<td>Controller: pass number of sticks to remove to interface player</td>
</tr>
<tr>
<td></td>
<td>Automated</td>
</tr>
<tr>
<td></td>
<td>View: display buttons for user to take 1, 2, or 3 sticks</td>
</tr>
<tr>
<td></td>
<td>Controller: listen to buttons for number of sticks to take</td>
</tr>
<tr>
<td></td>
<td>Controller: pass number of sticks to take to user player</td>
</tr>
<tr>
<td></td>
<td>View: display components for user to view game state</td>
</tr>
<tr>
<td></td>
<td>View: observe game to update state changes</td>
</tr>
<tr>
<td></td>
<td>View: query game state and write it to components, when game changes state</td>
</tr>
<tr>
<td></td>
<td>View: display text fields for user to view number of sticks last taken</td>
</tr>
<tr>
<td></td>
<td>View: observe interface player to update number of sticks last taken</td>
</tr>
<tr>
<td></td>
<td>View: query abstract class player for number of sticks last taken and write it to text field, when sticks are removed</td>
</tr>
<tr>
<td></td>
<td>View: display label for user to view number of sticks left in pile</td>
</tr>
<tr>
<td></td>
<td>View: observe game to update number of sticks left in pile</td>
</tr>
<tr>
<td></td>
<td>View: query game for number of sticks left in pile and write it to label, when sticks left in pile changes</td>
</tr>
<tr>
<td></td>
<td>View: observe game to report winner of game</td>
</tr>
<tr>
<td></td>
<td>View: query game for game is over or not, and if true, query game for winner of game and pass it to display option pane command</td>
</tr>
<tr>
<td></td>
<td>View: display option pane to report winner of game</td>
</tr>
</tbody>
</table>
6.8 Dependency Analysis of Nim Game

Design matrices of Nim Game were constructed to obtain a gestalt representation of dependencies within the application. Based on DESA design model (Figure 6.4), four matrices were constructed: UG-FR matrix, FR-DP<sub>m</sub> matrix, DP<sub>m</sub>-DP<sub>ui</sub> matrix, and DP<sub>ui</sub>-UA matrix. The implications of these matrices are discussed in the following paragraphs.

Figure 6.6 shows the UG-FR matrix of Nim Game. ‘X’ represents has mapping, ‘0’ represents no mapping, and a blank square represents inconsequential parent-child mapping. Absence of off-diagonal ‘X’ implies that user goals were mapped to functional requirements in a one-to-one mapping; there were no one-to-many mappings or many-to-one mappings. Therefore, the functional requirements did not cause any dependencies between the user goals, since each user goal was satisfied by an independent functional requirement. Such functional specification with a diagonal matrix is optimal, since it signifies a one-to-one relationship.

The FR-DP<sub>m</sub> matrix of Nim Game is similar to the UG-FR matrix shown in Figure 6.6, but the implications are different. In the FR-DP<sub>m</sub> matrix, an off-diagonal ‘X’ represents a dependency between two FRs caused by their DP<sub>m</sub>s. Two FRs are concluded to be dependent when modification of one of their DP<sub>m</sub>s affects the other DP<sub>m</sub>. For example, DP<sub>m</sub>2.1 and DP<sub>m</sub>2.2 have similar methods and data of concern, which is to determine number of sticks to take and command a Pile object to remove.
DESA: A Design Method that Reduces System Complexity

them (Table 6.2). Therefore, they are likely to share software code. If one class, Player, is used to contain these two similar methods, there will be no access restrictions between them, which will result in many cross-references. Modifying DP\textsubscript{m}2.1 will affect DP\textsubscript{m}2.2, and vice versa. Hence, FR2.1 and FR2.2 will be interdependent on each other, which are indicated by the two off-diagonal ‘X’s in Figure 6.7. The model is the source of this inter-dependency, not the user interface.

![Diagram](image)

**Figure 6.7.** The two off-diagonal ‘X’s imply that DP\textsubscript{m}3.1 and DP\textsubscript{m}3.2 cause an inter-dependency between FR2.1 and FR2.2

Since different classes, IndependentPlayer and InteractivePlayer, were used to contain the similar methods between DP\textsubscript{m}3.1 and DP\textsubscript{m}3.2, the inter-dependency is absent in the application (Figure 6.8). In fact, none of the DP\textsubscript{m}s causes dependencies between the FRs, which results in the full FR-DP\textsubscript{m} matrix being diagonal.

![Diagram](image)

**Figure 6.8.** Absence of off-diagonal ‘X’s imply that DP\textsubscript{m}3.1 and DP\textsubscript{m}3.2 do not cause any dependency between FR2.1 and FR2.2

The functional dependencies mentioned in the preceding paragraphs are different from client-server dependencies. A client object is dependent on a server object, because the former invokes the methods of the latter. For example, in DP\textsubscript{m}2.1, class IndependentPlayer is the client, while class Pile is the server, because an
IndependentPlayer object invokes the “remove” method of a Pile object and passes “sticks to take” as the argument. Hence, IndependentPlayer is dependent on Pile – IndependentPlayer’s code that invokes “remove” depends on how “remove” is specified in Pile. Such client-server dependency is not denoted in the design matrices. Nevertheless, other tools, such as the dependency structure matrix (Section 2.3; Steward, 1981; Eppinger, 1997), can be employed to analyze client-server dependencies (Sullivan et al., 2001; Sangal et al., 2005).

The DPm-DPui matrix has an implication similar to the FR-DPm matrix presented in the preceding paragraphs. In the DPm-DPui matrix, an off-diagonal ‘X’ represents a dependency between two FRs caused by their DPui’s – modification of one DPui affects the other DPui. For example, Nim Game has to display three types of game state information: sticks left in pile, sticks last taken by computer, and sticks last taken by user (Figure 6.5). Hence, the user interface subsystem has to display three almost identical panels on the graphical user interface, which contain the game state information. To avoid duplicate code when programming these panels, we can program one class Panel, and then create three instances of Panel during run-time. However, having to modify “sticks left in pile” panel implies that the other two panels will experience identical modification, and vice versa. This is undesirable because “display sticks last taken” and “display sticks left in pile” are different functions, FR3.1 and FR3.2 respectively (Table 6.1) – it is likely to have to modify one without changing the other. As a result, FR3.1 and FR3.2 are inter-dependent, which is represented by the two off-diagonal ‘X’s in Figure 6.9. The user interface is the source of this inter-dependency, not the model.

IndependentPlayer object invokes the “remove” method of a Pile object and passes “sticks to take” as the argument. Hence, IndependentPlayer is dependent on Pile – IndependentPlayer’s code that invokes “remove” depends on how “remove” is specified in Pile. Such client-server dependency is not denoted in the design matrices. Nevertheless, other tools, such as the dependency structure matrix (Section 2.3; Steward, 1981; Eppinger, 1997), can be employed to analyze client-server dependencies (Sullivan et al., 2001; Sangal et al., 2005).

The DPm-DPui matrix has an implication similar to the FR-DPm matrix presented in the preceding paragraphs. In the DPm-DPui matrix, an off-diagonal ‘X’ represents a dependency between two FRs caused by their DPui’s – modification of one DPui affects the other DPui. For example, Nim Game has to display three types of game state information: sticks left in pile, sticks last taken by computer, and sticks last taken by user (Figure 6.5). Hence, the user interface subsystem has to display three almost identical panels on the graphical user interface, which contain the game state information. To avoid duplicate code when programming these panels, we can program one class Panel, and then create three instances of Panel during run-time. However, having to modify “sticks left in pile” panel implies that the other two panels will experience identical modification, and vice versa. This is undesirable because “display sticks last taken” and “display sticks left in pile” are different functions, FR3.1 and FR3.2 respectively (Table 6.1) – it is likely to have to modify one without changing the other. As a result, FR3.1 and FR3.2 are inter-dependent, which is represented by the two off-diagonal ‘X’s in Figure 6.9. The user interface is the source of this inter-dependency, not the model.
This inter-dependency can be avoided by using a class ReportPanel to model the "sticks left in pile" panel, and a separate class PlayerPanel to model the other two panels (Figure 6.10). This is actually the design employed in the application. In reality, none of the DPui cause dependencies between the FRs, which result in the full DPm-DPui matrix being diagonal.

The DPui-UA matrix has implications which are different from the two preceding matrices, FR-DPm matrix and DPm-DPui matrix. An off-diagonal 'X' in the matrix represents a dependency between two UGs (user goals) caused by their UAs (user actions) – when users execute one of the UA, the other UA will be affected. This affects the users of the application, instead of the designers. Such source of dependency is more common among process control applications, where user interactions may be coupled (Vicente, 1999; Lo and Helander, 2007). The DPui-UA matrix of Nim Game is diagonal.

6.9 Conclusion of Case Study

DESA is effective in reducing the complexity of object-oriented software systems, as it minimizes the functional dependencies. Functional dependency can be caused by either the model subsystem or the user interface subsystem, or both, and DESA can locate the cause. Furthermore, DESA can aid object-oriented software designers to identify a suitable collection of classes for various software systems, and to allocate
appropriate responsibilities to the classes by using functional independence as the criterion.
CHAPTER 7. Conclusion

DESA (Design Equations for Systems Analysis) is an effective design method for reducing system complexity. Besides reducing the complexity of human-machine systems, it also reduces the complexity of object-oriented software systems. Software systems are known to be inherently complex.

DESA is fundamentally different from DSM (Design Structure Matrix). It reduces complexity by minimizing dependencies between system requirements, whereas DSM reduces complexity by managing the dependencies. Hence, DESA and DSM are able to complement each other – DSM can be employed to manage the remaining dependencies after they are minimized by DESA.

There are also differences between DESA and AD (Axiomatic Design). DESA utilizes a user-centered framework where user goals are ultimately mapped into user actions. This is absent in AD. Furthermore, DESA divides the physical domain of AD into model (internal structure) domain and user interface domain, which increases DESA’s effectiveness in reducing system complexity. This is because dependencies between system requirements can be caused by the model as well as the user interface. It is essential to locate the source of the dependencies to effectively remove them. Nevertheless, there are many similarities between DESA and AD, as DESA builds on AD. Both methods employ design equations to represent mapping between design domains, and design matrices to represent dependencies between system requirements.

DESA is also effective in designing usable products. It designs for effectiveness, efficiency, and satisfaction, which are the three elements of usability. This is achieved by the three rows of design domains in the DESA framework, where product parameters have to fulfill functional requirements, ergonomic requirements, and affective requirements of products. Besides being employed to increase the usability of human-machine systems, DESA can also be employed to increase the usability of tools and methods.
DESA can, thus, be employed when usability and simplicity are the major concerns in a product development process. The revised DESA framework and the established DESA terminology enable DESA to be employed systematically. Many of the terms, in the terminology, also appear in other design methods. Therefore, the terminology can serve as a guideline to distinguish common terms used in the product development field.

An overview of findings and problems solved in each chapter of this thesis is shown in Figure 7.1.

**Figure 7.1. Overview of findings and problems solved**
REFERENCES


DESQA: A Design Method that Reduces System Complexity


85


DESA: A Design Method that Reduces System Complexity


DESA: A Design Method that Reduces System Complexity


APPENDIX A. Formal Design Methods

A.1 Formal Design Methods for Designing Product Manufacturing Processes

A.1.1 SPC (Statistical Process Control)
Stapenhurst, 2005

Variation is inevitable in manufacturing processes. Nevertheless, variation has to be controlled, as excessive variation yields inconsistent products, and hence degrades quality. Figure A.1 shows process out of control and process in control.

(a) Process out of control
(b) Process in control

Figure A.1. Process out of control and process in control
DESA: A Design Method that Reduces System Complexity

SPC uses statistical techniques and control charts to control variation of manufacturing processes. In SPC, a process is assumed to be in control until evidence shows that the process central tendency and dispersion is off target. The central tendency is usually measured using the arithmetic mean, and dispersion is usually measured using the range or standard deviation. These control parameters are monitored using control charts (Figure A.2). The means, upper control limits, and lower control limits, of sample means and sample ranges, are calculated using statistical and derived formulas. Although SPC is capable of detecting excessive deviation, it is incapable of identifying causes of deviation.

![Control chart for means](image1)

![Control chart for ranges](image2)

Figure A.2. Control charts for means and ranges.

A.1.2 Acceptance Sampling
Stephens, 2001

Ideally, all units of a manufactured product should be inspected to evaluate their conformance to specified quality standards. However, there are many situations when 100% inspection is unfeasible: tight budget, insufficient time, or involvement of
destructive tests such as tensile tests. To solve this problem, Acceptance Sampling method was developed. In Acceptance Sampling, a sample of a population of products is evaluated to decide whether to accept or reject that population. There are two types of Acceptance Sampling: sampling by attributes and sampling by variables. The former is a more common type of sampling than the latter. In sampling by attributes, a sample is inspected by attributes – if the number of defectives is less than the pre-specified number, its population is accepted; otherwise the population is rejected. This is the most common method, and it has two types sampling plans: single sampling and multiple sampling (Figure A.3). A few techniques were developed to evaluate sampling plans: OC (operating characteristics) curve, AOQ (average outgoing quality), and ATI (average total inspection). Details of sampling plan evaluation can be found in Stephens (2001).

A.1.3 JIT (Just-in-time)
Monden, 1993

Organizations hold stocks of products, parts, and raw materials to cover short-term differences between their supply and demand. However, holding stocks requires much resource, because they occupy much space and require constant maintenance. JIT, which was developed to solve this problem, designs manufacturing processes such that there is no need to stock up. This is achieved by identifying causes of supply and demand differences and then taking actions to remove them. As a result, production would be more cost-effective, as there would be more efficient use of resources. This would inadvertently improve the quality of manufactured products. The following paragraphs present examples that demonstrate this method.

Operations are often more efficient with large batch sizes, as they reduce set-up cost and disruptions. But if demand is smaller, the big batches give stocks that are held for a long time. JIT looks for ways of reducing the batch size so that it closely matches demand.
Figure A.3. Single and multiple sampling plans

Figure A.4 shows components of production lead time. Suppliers often fix lead times that can be unnecessarily long. This reduces flexibility and encourages high stocks to cover uncertainty before another order can arrive. JIT looks for ways of continually reducing lead times. Monden (1993) proposed a framework for reducing lead time (Figure A.5).
When equipment breaks down, most organizations transfer resources and start making another product. JIT is based on continuous, uninterrupted production, so it does not allow this kind of flexibility. Hence, reasons for the breakdown should be identified, and measures taken to ensure that it does not happen again.
A.1.4 IDEF (Integrated DEFinition Language)
Yang and El-Haik, 2003

IDEF contains a family of methods developed to model manufacturing processes. IDEF0 method and IDEF3 method are developed for process mapping, and they are the most frequently used among many methods of process mapping. A process map is a schematic representation of a process. Figure A.6 shows the basic element of an IDEF0 process map. The box represents the manufacturing process, and the arrows represent items entering and leaving the process. IDEF0 process mapping involves multiple levels – Figure A.7 shows the first level process map of a PCB manufacturing process and the decomposed second level process map of sub-processes. This can be further decomposed into lower levels.

![Figure A.6. Basic element of an IDEF0 process map](image)

Although IDEF0 maps relationship of processes and sub-processes in detail, it does not illustrate temporal sequence of the process execution. IDEF3 was developed to solve this problem. It has two kinds of process sequence diagrams: process flow diagram and object state transition network diagram. Mayer and Painter (1991) reported IDEF family of methods in detail.
Figure A.7. First and second level process map of a PCB manufacturing process
A.1.5 Reliability Testing
Kececioglu, 1993

Reliability Testing can be employed at the testing and refinement stage of the product development process. It attempts to predict the reliability of products before production ramp-up. Product reliability can be defined as the probability of a product performing without failure under given conditions for a specified period of time. If reliability of a product is found to be low, the product and its manufacturing process have to be modified.

Product reliability is predicted using mathematical failure distributions. Many types of failure distributions are available, and Weibull distribution provides a useful fit in a significant number of cases. This is because it attains many shapes for various values of $\beta$, and hence, is able model a great variety of failure data. The Weibull probability density function is shown in Equation A.1.

$$f(T) = \frac{\beta}{\eta} \left( \frac{T - \gamma}{\eta} \right)^{\beta-1} e^{-(T-\gamma)^\beta}, \quad (A.1)$$

where $f(T) \geq 0$, $T \geq \gamma$, $\beta > 0$, $\eta > 0$, $-\infty < \gamma < \infty$

$\beta = \text{shape parameter}$

$\eta = \text{scale parameter}$

and $\gamma = \text{location parameter}$

**Table A.1. Times-to-failure data and the corresponding median rank**

<table>
<thead>
<tr>
<th>$j$</th>
<th>$T_j$</th>
<th>$\widehat{Q}(T_j) \times 100$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>6.70</td>
</tr>
<tr>
<td>2</td>
<td>34</td>
<td>16.23</td>
</tr>
<tr>
<td>3</td>
<td>53</td>
<td>25.86</td>
</tr>
<tr>
<td>4</td>
<td>75</td>
<td>35.51</td>
</tr>
<tr>
<td>5</td>
<td>93</td>
<td>45.17</td>
</tr>
<tr>
<td>6</td>
<td>120</td>
<td>54.83</td>
</tr>
</tbody>
</table>
Given a set of times-to-failure data and the corresponding median ranks, such as those displayed in Table A.1, $\beta$, $\eta$, and $\gamma$ can be obtained by plotting the data on a Weibull graph paper (Figure A.8). By substituting these parameters into $f(T)$, a probability density function that characterize the plotted data will be obtained. Equations A.2 and A.3 show how reliability, $R(T)$, can be calculated from $f(T)$.

\begin{align*}
R(T) &= 1 - F(T) \quad \text{(A.2)} \\
R(T) &= 1 - \int f(T) \, dT \quad \text{(integrate from 0 to } T) \quad \text{(A.3)}
\end{align*}
Figure A.8. Weibull plot of data in Table A.1 to obtain $\beta$, $\eta$, and $\gamma$
A.1.6 DFM (Design for Manufacturing)  
Ulrich and Eppinger, 2000

DFM is the most common method among the family of DFX (Design for X) methods. The X in DFX corresponds to quality characteristics such as reliability, robustness, serviceability, or manufacturability. DFM aims to simplify manufacturing process and lower manufacturing cost without sacrificing product quality. Ulrich and Eppinger (2000) proposed that DFM method has five steps:

1. Estimate the manufacturing costs  
2. Reduce the costs of components  
3. Reduce the costs of assembly  
4. Reduce the costs of supporting production  
5. Consider the impact of DFM decisions on other factors

Figure A.9 shows the flowchart of DFM. Manufacturing costs can be decomposed into component costs, assembly costs, and overhead costs, which can be further decomposed into lower levels (Figure A.10).
Proposed Design

Estimate the Manufacturing Costs

Reduce the Costs of Components
Reduce the Costs of Assembly
Reduce the Costs of Supporting Production

Consider the Impact of DFM Decisions on Other Factors

Recompute the Manufacturing Costs

N

Good Enough?

Y

Acceptable Design

Figure A.9. Flowchart of DFM
A.1.7 Six Sigma Process Improvement

Gupta, 2005

It is a traditional norm to control product variation such that products that are plus and minus three standard deviations from the mean fall within the specified range of acceptance. This results in 2,700 defectives per million products. Consequently, much resource is spent to fix these defects. Six Sigma Process Improvement is a quality improvement technique developed to solve this problem. It strives for near perfection in the consistency of manufactured products by reducing product variation such that products that are plus and minus six standard deviations from the mean fall within the specified range of acceptance (Figure A.11). This results in only 0.002 defects per million of products.
DES A: A Design Method that Reduces System Complexity

Besides reduction in defects, Six Sigma Process Improvement has two other benefits. First, it increases customer satisfaction. This is because a product that is closer to the target value yields more satisfaction. Second, it can afford minor shifts of product mean from the target value.

Six Sigma Process Improvement proposed a five-stage process to aid achievement of the Six Sigma goal: define, measure, analyze, improve, and control. This five-stage process was termed as DMAIC (Figure A.12). Formal manufacturing process design methods, such as those presented in preceding sub-sections, are employed at every stage. Statistical tools are also employed. DMAIC improves process falling below specification and looks for further incremental improvement. Although Six Sigma Process Improvement is developed to improve existing manufacturing processes, it can also be used to design new manufacturing processes.
A.2 Formal Design Methods for Designing Products

A.2.1 Design Optimization

Rao, 1996

Design Optimization can be employed at the system-level design stage and detail design stage of the product development process. It contains a family of methods such as FEM (Finite Element Method), Golden Section Method, Polynomial Approximation, Steepest Descent Method, Conjugate Gradient Method, Hooke and Jeeves Method, and Exterior Penalty Function Method. These methods aim to produce optimal designs, and an optimal design derives maximum benefit and
performance from a given amount of resource. Usage procedures of these methods are highly mathematical. To eliminate arduous optimization procedures, especially in complex designs, softwares were developed to automate some of these methods.

All design optimization problems can be expressed in a standard general form as follows:

Minimize objective function \( f(x) \), subject to:

\[
m \text{ number of inequality constraints } \quad g_j(x) \leq 0, \quad j = 1, 2, 3, \ldots, m \\
\text{l number of equality constraints } \quad h_k(x) = 0, \quad k = 1, 2, 3, \ldots, l
\]

where the \( n \) number of design variables is given by \( x_i \), or by the design variable vector

\[
\begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
\vdots \\
x_n
\end{bmatrix}
\]

Since Design Optimization is employed to maximize performance, maximize utilization of resource, and minimize cost, it designs for efficiency and cost-effectiveness.

**A.2.2 Design of Machine Elements**

Shigley et. al., 2004

Design of Machine Elements can be employed in the system-level design stage and detail design stage of the product development process. It guides selection of machine elements such as motor, chain, sprocket, belt, pulley, gear, shaft, and key. It aims to design for stability and strength by selecting machine elements with suitable attributes. Selection is based on detailed calculations using derived equations and physical laws. Each machine element has its own set of equations related to its attributes, and values of some variables in the equations can be read off from tables in design catalogues. Table A.2 shows offset sidebar chain dimensions extracted from a design catalogue.
A.2.3 DOE (Design of Experiments)

Condra, 2001

Experiments are often conducted to find out causes of an observed phenomenon or to find out correlation between variables. DOE is a method employed to ensure that these experiments are conducted in a systematic and objective way, so that valid and reliable results are obtained.

In general, experimenters begin DOE by specifying the objective of the experiment. This is followed by specifying the independent variables and dependent variables. Dependent variables are parameters that experimenters have to measure. Independent variables are conditions setup by experimenters. Terms such as factor and level are commonly used in independent variables. A factor is a design being studied in the experiment, and the levels of a factor are the values of the factor being examined. For
example, in an experiment to find out how several hull designs affect the speed of a motorboat, the factor is hull design, and there are 4 levels for hull design: design A, B, C, and D. The speed of the motorboat is then the dependent variable of the experiment.

Several strategies, such as planned grouping, randomization, and replication, are available to construct the structure of experiments. The number of factors and levels involved affects these strategies. Statistical tools, such as t-test and analysis of variance, are employed to analyze data collected from experimental runs. Condra (2001) explains these experimental structures and statistical tools in detail.

A.2.4 Function Analysis Diagram

Function Analysis Diagram can be employed in the planning stage, also known as the product definition stage, of the product development process. Function Analysis Diagram is highly similar to IDEF0 process mapping – both of them use boxes and arrows as their basic structure. In Function Analysis Diagram, the box represents a function, while the arrows represent inputs and outputs of the function (Figure A.13). Similar to IDEF0, each first level function can be decomposed into lower level sub-functions (Figure A.14). The main objective of using Function Analysis Diagram is to aid specify an exhaustive list of functions that is agreed by and clear to the whole design team. In short, this method designs for functionality of products.

![Figure A.13. First-level function analysis diagram](image-url)
A.2.5 Morphological Chart

Morphological Charts can be employed in the concept development stage, also known as the conceptual design stage, of the product development process. It tabulates the short-listed conceptualized solutions of each sub-function (Table A.3), which eases selection of conceptualized solutions. However, the chart itself does not guide designers on how to select the most suitable solution for each sub-function. Hence, Morphological Charts are usually employed together with concept selection matrices (Table A.4). This method does not design for any particular element of quality, as the design objectives depend on the selection criteria listed in the concept selection matrices.
### Table A.3. Morphological chart

<table>
<thead>
<tr>
<th>Probe Holders</th>
<th>X-axis (Table)</th>
<th>Y-axis (Table)</th>
<th>Z-axis (Table)</th>
<th>Pitch (Table)</th>
</tr>
</thead>
<tbody>
<tr>
<td>*Hinged Holder w Latch</td>
<td>*Shaft &amp; Bearing (friction pads)</td>
<td>*Electric Actuators</td>
<td>*Linked Pneumatic Legs with locking collars</td>
<td>*Ball &amp; Socket Joint</td>
</tr>
<tr>
<td>*2 Piece Holder w Knob screws</td>
<td>*Magnetic Chuck</td>
<td>*Rail &amp; Block System</td>
<td>*Gas Spring with Flanges</td>
<td>*Magnetic Chuck</td>
</tr>
<tr>
<td>*Mounted attachments with Latch</td>
<td></td>
<td></td>
<td></td>
<td>*Shaft &amp; Bearing (friction pads)</td>
</tr>
</tbody>
</table>
Table A.4. Concept selection matrices

(a) Concept screening matrix

<table>
<thead>
<tr>
<th>Selection Criteria</th>
<th>A Master Cylinder</th>
<th>B Rubber</th>
<th>C Ratchet</th>
<th>D (Reference)</th>
<th>E Swash Ring</th>
<th>F Lever</th>
<th>G Dial Screw</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ease of handling</td>
<td>0</td>
<td>−</td>
<td>−</td>
<td>0</td>
<td>0</td>
<td>−</td>
<td>0</td>
</tr>
<tr>
<td>Ease of use</td>
<td>0</td>
<td>−</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>−</td>
<td>0</td>
</tr>
<tr>
<td>Readability of settings</td>
<td>0</td>
<td>0</td>
<td>+</td>
<td>0</td>
<td>+</td>
<td>0</td>
<td>+</td>
</tr>
<tr>
<td>Dose metering accuracy</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>−</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Durability</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ease of manufacture</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Portability</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sum +'s</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Sum 0's</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Sum −'s</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Net Score</td>
<td>2</td>
<td>−1</td>
<td>−2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Rank</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Continue?</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Combine</td>
<td>Yes</td>
<td>Combine</td>
<td>Revise</td>
</tr>
</tbody>
</table>

(b) Concept scoring matrix

<table>
<thead>
<tr>
<th>Selection Criteria</th>
<th>Rating Weighted Score</th>
<th>DF Lever Stop Weighted Score</th>
<th>E Swash Ring Weighted Score</th>
<th>G Dial Screw Weighted Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ease of handling</td>
<td>3</td>
<td>0.15</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Ease of use</td>
<td>3</td>
<td>0.45</td>
<td>4</td>
<td>0.8</td>
</tr>
<tr>
<td>Readability of settings</td>
<td>3</td>
<td>0.25</td>
<td>5</td>
<td>0.5</td>
</tr>
<tr>
<td>Dose metering accuracy</td>
<td>3</td>
<td>0.75</td>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>Durability</td>
<td>2</td>
<td>0.3</td>
<td>4</td>
<td>0.6</td>
</tr>
<tr>
<td>Ease of manufacture</td>
<td>3</td>
<td>0.65</td>
<td>2</td>
<td>0.4</td>
</tr>
<tr>
<td>Portability</td>
<td>3</td>
<td>0.3</td>
<td>3</td>
<td>0.3</td>
</tr>
<tr>
<td>Total Score</td>
<td>2.75</td>
<td>3.45</td>
<td>3.10</td>
<td>3.05</td>
</tr>
<tr>
<td>Rank</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Continue?</td>
<td>No</td>
<td>Develop</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

A.2.6 SADT (Structured Analysis and Design Technique)
Marca and McGowan, 1988

SADT has several phases: analysis, design, implementation, integration, testing, installation, and operation. However, the main focus of this method is in the analysis phase and design phase, in which SADT Activity Diagram is employed. This diagram
is a structured approach towards requirement specification and solution conceptualization, and hence, can be employed in the planning stage and concept development stage of the product development process (Ross and Schoman, 1977). Like the Function Analysis Diagram, which also aids requirement specification, SADT Activity Diagram has a basic structure of boxes and arrows, and its first-level requirements are decomposed into lower level requirements. However, there are two main differences between these two diagrams:

1. SADT Activity Diagram aids specification of all kinds of requirements, but Function Analysis Diagram aids specification of functional requirements only
2. The arrows in SADT Activity Diagram represent relational constraints between the requirements, while the arrows in Functional Analysis Diagram represent inputs and outputs of functional requirements.

Figure A.15 shows first-level and second-level of the SADT Activity Diagram. This method designs for functionality and effectiveness in products.
DES A: A Design Method that Reduces System Complexity

(b) Second level activity diagram

Figure A.15. First-level and second-level of the SADT Activity Diagram

A.2.7 QFD (Quality Function Deployment)
Franceschini, 2002

Figure A.16. Customer attributes, CTSs, functional requirements, design parameters, and process variables in QFD
QFD can be employed in the planning stage and concept development stage of the product development process. Initially developed to analyze customer attributes and design parameters, it can also be used to analyze CTSs (critical to satisfaction attributes), functional requirements, and process variables (Figure A.16). A commonly used tool in QFD is the "house of quality". Figure A.17 displays the "house of quality" of customer requirements versus design parameters, which contains the following information:

- List of customer requirements
- List of design parameters
- Relationship between customer requirements and design parameters
- Comparison with competitors' products
- Importance rating of design parameters
- Correlation between every pair of design requirements
- Customers' rating of customer requirements

![Figure A.17. “House of quality” of customer requirements versus design parameters](Image)
This information aids making of cost-effective decisions when adjusting product
attributes to improve the product. This distinguishes QFD from other requirement
specification and/or solution conceptualization methods. Hence, on top of designing
for functionality and effectiveness, QFD also designs for cost-effectiveness. QFD and
“house of quality” are clearly presented by Hauser and Clausing (1988).

A.2.8 TRIZ (Theory of Inventive Problem Solving)
Terninko et. al., 1996

TRIZ can be employed in the planning stage and concept development stage of the
product development process. This method was developed based on a review of
numerous successful patents. The review shows that the success of an invention
depends on how well contradiction between features of the invention is resolved. This
is similar to the concept of coupling in Axiomatic Design. TRIZ has four phases in its
problem solving process: problem definition, problem classification and tool selection,
problem solution, and solution evaluation.
In the problem definition phase, functional modeling and functional analysis is
conducted. This is followed by an evaluation of the ideality of the current system
(Equation A.4).

\[
\text{Ideality} = \frac{\Sigma \text{benefits}}{\Sigma \text{costs} + \Sigma \text{harm}}, \quad (A.4)
\]

Where \(\Sigma \text{benefits}\) = sum of the values of system’s useful functions
\(\Sigma \text{costs}\) = sum of the expenses for system’s performance
\(\Sigma \text{harm}\) = sum of “harms” created by harmful functions

In the problem classification and tool selection phase, problems are identified and
then classified into the following categories: physical contradiction, technical
contradiction, imperfect functional structure, excessive complexity, inadequate system
improvement, and unable to develop useful functions. TRIZ has many methods to
resolve the problems in every problem category. Hence, many alternative solutions
can be generated in the solution generation phase. The best solution is selected in the
concept evaluation phase using concept evaluation methods. External concept evaluation methods are usually employed. For example, Axiomatic Design is employed to evaluate the complexity of various solutions. TRIZ designs for functionality, simplicity, and cost-effectiveness.

**A.2.9 FMEA (Failure Mode and Effect Analysis)**

Yang and El-Haik, 2003

FMEA designs for safety and workability. It is employed to assess potential risks in designs. It has nine steps in its assessment procedure, which is documented in an FMEA worksheet (Table A.5):

1. Identify the potential failure modes by asking what can go wrong
2. Specify the potential failure effects of each potential failure mode by asking what are the consequences if things go wrong
3. Rate the SEV (severity) of each potential failure effect using a scale of 1 (lowest) to 10 (highest)
4. Identify the potential causes of each potential failure by asking why would it go wrong
5. Rate the OCC (likelihood of occurrence) of each potential cause using a scale of 1 (lowest) to 10 (highest)
6. Specify the current controls by asking what are the existing prevention measures
7. Rate the DET (likelihood of detection) of potential failure by each current control using a scale of 1 (highest) to 10 (lowest)
8. Calculate the RPN (Risk Priority Number) of each potential failure (Equation A.5)

\[ RPN = SEV \times OCC \times DET \]  

(A.5)

9. Specify recommended actions for potential failures that have high RPN

This procedure is repeated until satisfactory RPN is achieved. Guidelines to aid rating of SEV, OCC, and DET can be found in rating tables developed by different industries. Besides being employed in product design, FMEA can also be employed in manufacturing process design. The former is termed as Design FMEA, while the latter
is termed as Process FMEA. For early detection of potential failures, FMEA should be employed in the concept development stage of the product development process.

Table A.5. Design FMEA worksheet

<table>
<thead>
<tr>
<th>Item/Function</th>
<th>Potential Failure Mode(s)</th>
<th>Potential Effect(s) of Failure</th>
<th>Potential Cause(s)/Mechanism(s) of Failure</th>
<th>Design Lead</th>
<th>Revision Date</th>
</tr>
</thead>
</table>

A.2.10 RD (Robust Design)

Fowlkes and Creveling, 1995

RD can be employed in the concept development stage, system-level design stage, and detail design stage of the product development process. It aims to design products that are insensitive to variation so that consistency is easier to achieve when the products are manufactured. The following is an example to demonstrate how this can be done.

Figure A.18 shows two designs of a 100m³ cylinder. The volume of the manufactured cylinder has to be highly consistent. The following calculation, based on Equation A.6, evaluates which design will allow easier achievement of consistency:

\[ V = \pi r^2 h \quad \partial V = 2\pi r h \partial r \quad (A.6) \]

Design A:

\[ V \pm \partial V = \pi (2.523)^2 (5) \pm 2\pi (2.523)(5) \partial r \]

\[ = 100 \pm 79 \partial r \]
DES A: A Design Method that Reduces System Complexity

Design B: \[ V \pm \partial V = \pi (3.257)^2 (3) \pm 2\pi (3.257) (3) \partial r \]
\[ = 100 \pm 61 \partial r \]

During manufacturing of the cylinder, \( \partial r \) (variation of radius) will be reduced to the minimal to minimize \( \partial V \) (variation of volume). However, for the same \( \partial r \), design B will have a lower \( \partial V \) than design A. Hence, design B allows easier achievement of consistency during manufacturing.

Sources of variation, such as \( \partial r \) in the above example, can be termed as noise. The fundamental concept of RD is to identify the noise, identify the factor interacting with noise \( (2\pi r h) \), and exploit the factor to minimize sensitivity to noise. Therefore, RD designs for robustness. More techniques of RD can be found in Taguchi’s Robust Parameter Design.

Fowlkes and Creveling (1995) shows that RD aims for a two-step optimization process (Figure A.19):

1. Reduce variation
2. Shift mean to target value
The optimization process has to follow this sequence because the mean will shift when reducing variation.

![Diagram of two-step optimization process of RD](image)

Figure A.19. Two-step optimization process of RD

A.2.11 DFSS (Design for Six Sigma)
Yang and El-Haik, 2003

Although Six Sigma Process Improvement is capable of improving existing manufacturing process to increase consistency of its manufactured products, the increase is limited because the products might have designs that cause difficulty to achieve consistency. DFSS was developed to solve this problem. It designs products for consistency, and is employed early in the product development process – from
planning stage to detail design stage. Besides designing for consistency, it also
designs for functionality, effectiveness, robustness, simplicity, and cost-effectiveness.

DFSS has four phases: identify requirements, characterize the design, optimize the
design, and verify the design. These four phases are sometimes termed as ICOV.
Design methods, such as those presented in the preceding sub-sections, are employed
in each phase. For example, QFD can be employed in phase 1 to identify customer
requirements and product functions. AD can be employed in phase 2 to conceptualize
design parameters that are uncoupled, so that tuning them to their targeted values is
easier during manufacturing. RD can be employed in phase 3 to make the parameters
insensitive to variation so that high consistency is easier to achieve during
manufacturing. Therefore, DFSS enhances the Six Sigma strategy of Six Sigma
Process Improvement, and it also ensures all desired functions and requirements are
delivered.

A.2.12 MUSE (Method for Usability Engineering)
Lim and Long, 1994

Lim and Long (1994) proposed that for a usability design method to be effectively
employed, it needs to have the following requirements:

- Explicit definition of its scope, product, and process
- Development of its notations to support specific design descriptions
- Explicitly locate it against a structured design framework
- Development of computer-based tool after specification of a reasonably complete
  structured method

Lim and Long developed a usability design method that satisfies all the above
requirements. This method was termed as MUSE. It can be employed at the planning
stage and concept development stage of the product development process. The
structured framework of MUSE has three phases and a few stages within each phase
(Figure A.20):
1. Information elicitation and analysis phase: extant systems analysis stage and generalized task model stage

2. Design synthesis phase: statement of user needs stage, composite task model stage, and system and user task model stage

3. User interface specification phase: interaction task model stage, interface model stage, and display design stage

Figure A.20. The structured framework of MUSE

In the extant systems analysis stage, extant systems are analyzed to understand user needs and problems. A generalized task model of the extant system is then constructed in the generalized task model stage.

With the background information concluded in the preceding two stages, designers proceed to the statement of user needs stage to specify the requirements of the target system. This is followed by the composite task model stage, where the constructed
generalized task model is transformed into a composite task model of the target system. The components in a composite task model can be segregated into on-line tasks (i.e. computer-supported tasks) and off-line tasks (i.e. manual tasks). In the system and user task model stage, the on-line task portion of the composite task model is decomposed into a system task model, while the off-line task portion is decomposed into a user task model.

In the interaction task model stage, the system task model from the preceding stage is further decomposed into an interaction task model. An interaction task model represents human-computer interactions required to achieve on-line task goals of the target system. Designers have to iterate between interface model stage and display design stage to ensure design coherence. The former aims to conceptualize the semantics and operational characteristics of interface objects while the latter aims to conceptualize the spatial location and layout of the objects.

MUSE designs for functionality, effectiveness, and usability.