Design and Implementation
of an Intelligent Ideation Database

by

Kai Wei Hong

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 27, 1997

Copyright 1997 Kai Wei Hong. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce
and to distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author

Department of Electrical Engineering and Computer Science
May 27, 1997

Certified by

Nam P. Suh
Ralph E & Eloise F Cross Professor and
Mechanical Engineering Department Head
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Design and Implementation of an Intelligent Ideation Database

by
Kai Wei Hong

Submitted to the
Department of Electrical Engineering and Computer Science

May 27, 1997

In Partial Fulfillment of the Requirements of the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

By applying the axiomatic design method, a knowledge representation is created to support the storage, retrieval, and query of design objects. The Ideation Database is built around the representation. Its purpose is to generate possible design solutions, analyze those proposed solutions, and inform the designer of the results. The application architecture and user interface are presented, along with a case study to demonstrate the use of the knowledge representation.

The Intelligent Ideation Database enables designers using the AD method to utilize the knowledge of others in the design process. The goal is to prevent redundant design work and aid in the creative process by allowing designers to augment their own body of knowledge with the knowledge of others. As the database is updated with more designs, it becomes an even more effective tool in expanding the human design intellect.

Thesis Supervisor: Nam P. Suh
Title: Ralph E & Eloise F Cross Professor and Mechanical Engineering Department Head
Acknowledgments

The author acknowledges Professor Nam P. Suh for his patience and guidance throughout the thesis project. His advice and example were invaluable to the author throughout the year.

The author acknowledges the Silicon Valley Group, Inc. for funding this thesis work as part of their own-going support for research in the field of design.

The author acknowledges Peter Filosi, Michael Creighton, David Morris, and Ernest Frazer of SVG Lithography Systems, Inc. for providing the information and assistance needed to construct a case study with which to test the system.

The author acknowledges Vigain Harutunian and Derrick Tate. Their contributions helped the author gain a better perspective on axiomatic design and the design process as a whole.

The author acknowledges Shaun Abrahamson. His feedback helped to formalize a great jumble of ideas. Also, by breaking his collarbone in a freak snowboarding accident, he convinced the author to spend less time snowboarding and more time working on his thesis.

The author acknowledges his family for their support through five years of schooling which culminates with the completion of this master’s thesis. Their love and encouragement convinced the author he could accomplish anything he put his mind to.
Dedication

To my father and sister

and to my mother in heaven

“[Love] always protects, always trusts, always hopes, always perseveres...”
1 Corinthians 13:7
# Table of Contents

The Design and Development of an Intelligent Ideation Database .......... 3  
Abstract ................................................................................. 3  
Acknowledgments ................................................................. 5  
Dedication ................................................................................. 7  
Table of Contents ..................................................................... 9  
List of Figures ........................................................................ 13  
List of Tables ............................................................................ 15  

1. Introduction ........................................................................ 17  
  1.1 Motivation ........................................................................ 18  
  1.2 Thesis Scope ....................................................................... 20  
  1.3 Thesis Overview ............................................................... 21  

2. Review of Research in Computer Aided Design Tools ................. 23  
  2.1 CANDLE Product Modeling Language ............................... 23  
  2.2 Modeling Information in Design Background .................... 24  
  2.3 An Intelligent Object Oriented Approach to Design and Assembly ... 25  
  2.4 Knowledge Acquisition Method for Conceptual Design ............ 26  
  2.5 Remarks ........................................................................... 27  

3. Overview of Axiomatic Design .............................................. 29  
  3.1 Domains ........................................................................... 30  
  3.2 Functional Requirements ................................................... 31  
  3.3 Design Parameters ............................................................ 32  
  3.4 Hierarchical Decomposition .............................................. 33  
  3.5 Zigzagging ......................................................................... 34  
  3.6 Design Matrix .................................................................... 35  
  3.7 Independence Axiom .......................................................... 37  
  3.8 Information Axiom .............................................................. 38  
  3.9 Remarks ........................................................................... 40  

4. Thinking Design Machine ...................................................... 41  
  4.1 TDM Concept ..................................................................... 41  
  4.2 Architecture of the TDM ..................................................... 43  
  4.3 Database for the TDM ....................................................... 44
List of Figures

Figure 3-1. Design domains ........................................ 30
Figure 3-2. Hierarchical decomposition and mapping .................. 33
Figure 3-3. Decomposition by zigzagging ........................... 35
Figure 3-4. Examples of the three types of design matrices ............... 37
Figure 3-5. Probability distribution functions for FRs and DPs ............. 39
Figure 4-1. Schematic diagram of the TDM ............................. 42
Figure 5-1. Relationships between FR and DP objects ................... 47
Figure 5-2. Interaction model between FR and DP objects ................. 52
Figure 6-1. System hierarchy for the Ideation Database .................. 59
Figure 7-1. Ideation database application structure ........................ 61
Figure 7-2. Analysis View form ...................................... 62
Figure 7-3. FR selection dialog ....................................... 63
Figure 7-4. Database View form ...................................... 65
Figure 7-5. Design parameter view dialog ............................... 66
Figure 7-6. Attribute dialog ......................................... 67
Figure 7-7. Search dialog ........................................ 68
Figure 8-1. Implicit relationships in Microsoft Access .................... 71
Figure 9-1. Cathode tip erosion ....................................... 78
List of Tables

Table 3-1. Analogy among different design problems ................................. 31
Table 5-1. Data fields for FR and DP objects ........................................... 48
Table 5-2. Examples of state field values .................................................. 49
Table 8-1. Example of the recursive algorithm .......................................... 74
Table 9-1. Top level FR for lamp project .................................................. 80
Table 9-2. Top level DP for lamp project .................................................. 80
Table 9-3. Second level FRs for lamp project ............................................ 81
Table 9-4. Second level DPs for lamp project ............................................ 81
Chapter 1. Introduction

Design has long been considered more in the realm of art than science. Solutions are often obtained through the application of equal measures of creativity and trial-and-error prototyping. Once obtained and realized in either physical form or simulation form, they are judged subjectively by how they work and whether they satisfy some measurable criteria such as energy efficiency or speed. The good designs also tend to have an aesthetic quality to them that is hard to quantify. However, without an objective approach to guide us, it is difficult to determine and teach good design practice. Experience becomes the only tool that designers can call upon to help them in their attempts to solve new problems. It is also the only way to discern the good designs from the bad ones. It was in this type of environment that axiomatic design was born.

Axiomatic design is a method developed by Professor Nam P. Suh at the Massachusetts Institute of Technology. The AD method provides a set of principles that can assist the design engineer in identifying the good designs from an infinite number of possibilities. There are two main design axioms: the Independence Axiom and the Information Axiom. These axioms are conceptually simple but enable the designer to answer several fundamental questions such as: “Is this a good design?” and “Why is this design better than others?” [12]. Chapter 3 of this thesis explores the axiomatic design method more fully.
Once axiomatic design provides the appropriate framework for design documentation and development, software systems can be developed to assist the designer in the process. The goal of this thesis is to design and develop an Intelligent Ideation Database system which utilizes the axiomatic design method to represent, synthesize, and evaluate design data more efficiently and more thoroughly than is possible manually. This research explores part of the concept outlined by Suh in the Thinking Design Machine (TDM). The TDM is a conceptual model of an intelligent machine that is capable of automatically generating designs which are superior to those currently possible in several key aspects such as robustness and ease of manufacture. The Ideation Database is an important part of the TDM and is the focus of the thesis.

1.1 Motivation for a Design Database

Design relies a great deal upon experience. The more experience a particular designer or design team has, the broader the range of possible solutions. As the use of computers has spread, designers have been eager to utilize the processing power available to assist them in their pursuit of good design and in their efforts to capture design experience. Many CAD (computer aided design) applications have been developed for this purpose. However, those tools tend to focus on the later stages of the design process, such as modeling and drawing. Also, the knowledge transfer with CAD is mainly from human to computer.

As useful as this type of CAD tools has been and continues to be, the focus in the development of new design tools is shifting to the earlier stages of the design process.
The most critical decisions regarding the design are made during these first few stages. However, before effective software systems can be developed, the conceptual design process must be systematized for machine processing [15]. In order to accomplish this, the axiomatic design process is applied. Then, as with human designers, the machine designer relies on the body of knowledge it contains to assist it in generating solutions. It is the database that maintains the necessary knowledge.

The database provides the framework for a large collaborative environment in which designers can share their experience. The designers access this collective pool of knowledge and experience and thereby augment their own expertise during the design process. A well-designed database system prevents a lot of duplicate work by allowing designers to view information about components used in previous designs without having to go through the process of designing those components themselves. They can then choose to incorporate those elements into their own designs or construct entirely new elements. New elements enter the database for the benefit of future designers. In creating new elements, the designer benefit from the knowledge gained from consulting the database. The more design knowledge the database contains, the more effective it becomes.

The thesis project seeks to assist the designer by developing an efficient and easy-to-use database of design elements. Such a database enables designers to search a repository of pre-existing design elements for components they can use in their designs. Also, if given a set of requirements, it is able to suggest possible design solutions based on its state.
1.2 Thesis Scope

The design and development of the entire Thinking Design Machine is a daunting task. As such, the Axiomatic Design Group at M. I. T. has chosen to break the problem up into modules. The first generation of work on the axiomatic design software was done by Vigain Harutunian and Derrick Tate under the supervision of Professor Suh. They developed a software system which allows the user to document a design using the AD method. The software supports zigzagging, hierarchical decomposition, and matrix manipulation. Lately, they have also added several graphical elements to the tool, such as a module-junction diagram and a flow chart view of the design.

With that work in place, this thesis project focuses on the design and implementation of the Ideation Database system. The database is developed separately from the existing AD software to yield the maximum amount of flexibility in implementation. The analysis portion of the database deals exclusively with the selection of appropriate design parameters given a set of user specified functional requirements. Hierarchical decomposition and zigzagging are not implemented since it is intended that both the existing AD software and the new Ideation Database system be used in concert. The designer documents the design using the AD software and then refers to the Ideation Database once he has defined the functional requirements at a particular level of decomposition. The designer enters the functional requirements into the Ideation Database analysis form and directs the system to suggest possible design solutions for those requirements. Once the designer is satisfied with the design parameters proposed
by the database for the functional requirements, he returns to the AD software and enters the information.

1.3 Thesis Overview

The approach of this thesis is to start by building a frame of reference from background material before presenting the implementation of the Ideation Database. It begins with a review of the state of the art in computer aided design systems and a discussion of the axiomatic design method and the TDM. Once the baseline for assessment has been built, the thesis presents the various components of the database system. The following list summarizes the content of each chapter.

- Chapter 2 presents a review of current research in the field of computer aided design systems. It is meant as a primer on the issues involved in the field and as basis upon which to judge the method selected in the thesis work.

- Chapter 3 provides an overview of the axiomatic design method. The topics covered include the concept of domains, functional requirements, design parameters, hierarchical decomposition, zigzagging, and the design matrix.

- Chapter 4 describes the concept of the Thinking Design Machine. It presents the logical structure and focuses on the database component of the TDM.

- Chapter 5 presents the knowledge representation developed for the Ideation Database. The structure and fields of the representation are discussed. It is then compared to the representation presented by the TDM.
• Chapter 6 applies the axiomatic design method to the Ideation Database system. The top level functional requirements and design parameters are determined and then decomposed.

• Chapter 7 describes the user interface of the Intelligent Ideation Database more fully. It presents the two main modules in the user interface, the Analysis View and the Database View and documents their use.

• Chapter 8 focuses on the some interesting portions of the implementation such as the maintenance of relationships in Access and the logic used to generate the DP combinations.

• Chapter 9 presents a case study of the lamp life improvement project which was used to test the software and the knowledge representation.

• Chapter 10 summarizes the thesis work and proposes future work towards fulfilling the TDM concept.
Chapter 2. Review of Research in Computer Aided Design Tools

A great deal of research work is currently ongoing in the field of computer aided design tools. Researchers are utilizing the increasing computational power of newer and faster machines to develop more and more sophisticated software systems to assist the designer. They are also utilizing many new technologies from the field of computer science, such as expert systems, inference nets, and object oriented programming. This research work separates into two categories of focus: knowledge representation and design automation. This chapter covers some work in both categories, but is by no means intended to be a comprehensive overview of the field. The work of others is discussed here to provide a frame of reference from which to assess the work presented in this thesis.

2.1 CANDLE Product Modeling Language

Before much work can be done in automating the design process, a modeling language must be developed to represent conceptual design work. That was the motivation behind the CANDLE modeling language developed by a team from the Royal Institute of Technology in Stockholm, Sweden. Their approach enabled the use of engineering terminology for modeling early design work. By expanding this basic
lexicon with physical and solution principles for specific topics, they were able to support a wider range of design tasks.

They chose to follow the model of the design process described by Pahl and Beitz in *Engineering Design - A Systematic Approach* for its pragmatism and ease of use. The different design phases laid out by Pahl and Beitz resemble the first two steps of the TDM concept defined by Suh [15]. By using this structure, their syntactic and semantic framework, and some graph theory, the Swedish researchers developed a language which quantified the description of conceptual design models [1]. However, their system did not address the issue of verifying that a proposed solution actually satisfied the design requirements (this was proposed as future work). Also, by requiring domain specific language extensions for each class of design problem, CANDLE introduced a significant amount of overhead associated with extending the system.

2.2 Modeling Information in Design Background

The work of Suzuki’s team from the University of Tokyo also concentrated on capturing design information albeit a different type. The team looked at the design environment and realized that most of the background information generated during the design and development process was not being represented or stored in a useful manner. Based on their evaluation of a prototype system, they determined that the design background information (DBI) should be an important element of product data management systems. Since the boundary between foreground and background information is seldom clear, they assumed the only informational elements of the design process in the foreground were the drawings and models. All other informational
elements were considered background and in the domain of the DBI. This includes
design requirements, intents, methodologies, standards, design histories, and records.

Through case studies and discussions with designers, the DBI team concluded that
by handling background information the collaborative work of designers could be more
efficient and that design performance could be improved by using DBI from design
examples [16]. Through their conclusions, the University of Tokyo team verified the
importance of modeling design information. By applying the axiomatic design method to
the design process and developing a flexible knowledge representation, this thesis project
incorporates some of the design information Suzuki’s team found necessary, such as
design requirements and intents (which are both captured by the FRs of a design), into an
interactive framework to assist designers in the generation of good designs. Their work
re-emphasizes the importance of capturing design information in a database.

2.3 An Intelligent Object Oriented Approach to Design and Assembly

Ishmail’s research group from the University of Liverpool focused more on the
analysis portion of design rather than concept definition and development. The system
they built was a prototype of an intelligent CAD system to support the design of
progressive blanking dies. Their system accepted as inputs geometry, stock strip
specifications, machine specifications, and other design and operational considerations.
It returned a detailed design, all press tool part drawings, and a bill of materials. The
software system consisted of a Kappa expert system shell linked to an extended
AutoCAD interface.
By utilizing the rule-based expert system provided by Kappa, the researchers were able to build a system which could check the feasibility of the user’s conceptual design and advise the designer of the unreasonable elements of his design. They believed that it was not difficult, in principle, to extend their system into a complete intelligent progressive die design system. This could be accomplished by coding generic rules into the expert system [7].

This thesis project seeks to provide the same functionality sought by the Liverpool researchers, namely automation of the design process, to the general design process. Rather than focusing on part geometries and other constraint-level details, the thesis and the underlying axiomatic design method hopes to help the designer in the more fundamental task of determining “why” do something versus “how” to do it.

2.4 Knowledge Acquisition Method for Conceptual Design

Research by Kawakami’s group in Japan proposed a method for acquiring conceptual design knowledge in physical systems. Following the method, the structural features of a design were analyzed to yield a description of how they functioned, how they reached the design goals, and why those features were used in the first place. The method was based on explanation-based learning (EBL), value engineering methodologies, and axiomatic design approaches. By examining “how” the structural features functioned, a generalized functional diagram was obtained and examined. This enabled the group to extract various levels of general design knowledge. By examining “why” those structures were chosen, a deeper understanding into those structures arose.
The answers to “why” became evident by applying an axiomatic approach to the design objects in question.

The group used hierarchical modeling and functional analysis to create a knowledge representation that could be analyzed by a software system. They proposed to generalize the set of rules obtained through modeling and analysis into a semantic network of goal concepts (GCs). This network would then be applicable to a broader range of design problems. The set of GCs defined “how” a structural element functioned. Once the GCs were obtained, axiomatic design was used to examine “why” a structural element was chosen. They defined a direct one-to-one relationship between GCs and functional requirements (FRs) often using the terms interchangeably. The entire network would then act as the input to the EBL [8].

The approach followed by Kawakami’s team provides a good structure for representing general design knowledge and particularly physical concepts. Future work in an axiomatic design database system can incorporate their framework into the storage of design knowledge. It provides a more formalistic approach to the definition of functional requirements which ultimately would lead to more efficient analysis by a software system.

2.5 Remarks

By providing a glimpse at some of the work being done in the field of computer aided design systems, the need for efficient knowledge representations becomes evident. The structure and function of the database directly bounds the effectiveness of any
analysis done by a software system. It is this fact that frames the work undertaken in this thesis.
Chapter 3. Overview of Axiomatic Design

Professor Nam P. Suh and his colleagues at MIT formulated axiomatic design in the mid 1970's as part of an effort to transform the field of design and manufacturing into an academic discipline. The foundations of axiomatic design were later presented in a book published in 1990 [12]. Additionally, many papers have been written examining the axiomatic approach and its impact in many fields. Gebala and Suh applied the approach to the design of artificial skin [3]. Papers by Kim, et al., and Do and Park used the axiomatic design method in the design of software systems [9, 2]. Nordlund, et al., presented an overview of the growth in the use of axiomatic design in industry [11].

Axiomatic design incorporates several concepts into a unified framework for design. The concepts of design domains, hierarchical decomposition, and zigzagging are joined to those of functional requirements, design parameters, and the design matrix. Together, these concepts provide the structure needed to improve the design process by providing a theoretical foundation upon which to approach the art of design. By providing logical and rational thought processes and tools, axiomatic design seeks to make human designers more creative and the fruits of their labor more robust and effective than was possible in the past.
3.1 Domains

Axiomatic design prescribes the use of various *domains* in the design world: the *customer domain* \{Customer Needs, CNs\}, the *functional domain* \{Functional Requirements, FRs\}, the *physical domain* \{Design Parameters, DPs\}, and the *process domain* \{Physical Variables, PVs\}. These four domains and their relationships are shown below.

The domain on the left represents "what we want to achieve" relative to the domain on the right which represents "how we are going to achieve it." For example, a "what" in the customer domain is satisfied by an associated "how" in the functional domain. The mapping from "what" to "how" is governed by the Independence Axiom which requires the designer to maintain the independence of the requirements at all times during the design process. All designs fit into these four domains [13].

The customer domain contains the customer needs or attributes the customer is looking for in a product or process. In the functional domain, those customer needs are
formulated in terms of functional requirements (FRs) and constraints (Cs). In order to satisfy those requirements, a set of design parameters (DPs) is devised in the physical domain. Lastly, to produce the product specified by the design parameters, the designer or process engineer develops a process characterized by a set of process variables (PVs). Examples of the domain elements are presented in the following table.

<table>
<thead>
<tr>
<th>Domain Character Vectors</th>
<th>Customer domain {CNs}</th>
<th>Functional domain {FRs}</th>
<th>Physical domain {DPs}</th>
<th>Process domain {PVs}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manufacturing</td>
<td>Attributes which customers desire</td>
<td>Functional requirements specified for the product</td>
<td>Physical variables which can satisfy the functional requirements</td>
<td>Process variables that can control the design parameters</td>
</tr>
<tr>
<td>Software</td>
<td>Attributes desired in software</td>
<td>Output specification of program codes</td>
<td>Input variables, algorithms, modules, or program codes</td>
<td>Sub-routines, machine codes, compilers, or modules</td>
</tr>
<tr>
<td>Organization</td>
<td>Customer satisfaction</td>
<td>Functions of the organization</td>
<td>Programs or offices</td>
<td>People and other resources that can support the programs</td>
</tr>
<tr>
<td>Systems</td>
<td>Attributes desired of the overall system</td>
<td>Functional requirements of the system</td>
<td>Machines, components, or sub-components</td>
<td>Resources (human, financial, materials, etc.)</td>
</tr>
</tbody>
</table>

Table 3-1. Analogy among different design problems [14]

3.2 Functional Requirements (FRs)

Functional requirements are defined as the minimum set of independent requirements that represent the design goals, subject to constraints, stated in a solution-neutral manner. Examples of FRs are “move block”, “lift weight”, and “provide power.” As in the examples, the description string for an FR starts with a verb.
The design process begins with the definition of an appropriate set of FRs in response to a set of customer needs. The process ends with the development of an object which satisfies the FRs. The definition of FRs for a particular set of customer needs is highly subjective. Each designer utilizes his own experience to determine how the customer needs should be addressed in the functional domain. That is how variation is introduced into the design process and how one set of customer needs can yield many different sets of functional requirements. For example, the customer need “transportation” can yield FRs ranging from “move from point A to point B” which is active to “get from point A to point B” which is passive. This variety is evidenced by the several different designs available for any particular product made today.

3.3 Design Parameters (DPs)

Design parameters are defined as the physical embodiments or objects chosen by the designer to satisfy the functional requirements in the physical domain. Examples of DPs are “gear”, “wheel”, and “engine.” The description string for a DP centers on a noun. As with FRs, the determination of a set of DPs for a corresponding set of FRs is largely up to the discretion of the designer. Given the potentially infinite set of DPs for each set of FRs, the axiomatic design framework provides the design axioms to determine which of those sets constitute better designs and why. The design axioms are discussed in detail below.
3.4 Hierarchical Decomposition

Another property of the axiomatic design method is the hierarchical nature of the design domains. The design process moves from a highly-abstracted system level (one CV to one FR to one DP to one PV) to levels of increasing detail until the finest granularity necessary is reached. The nodes at lowest level in the hierarchy are called the leaf nodes.

There is a one-to-one relationship between all elements in the hierarchy. Each CV has an associated FR and each FR has an associated DP. That DP has an associated PV. The one-to-one correlation between elements is necessary to maintain both simplicity (the minimum number of elements are used) and independence (a system with fewer “how”s then “what”s cannot satisfy the Independence Axiom). The figure below illustrates the hierarchy and mapping.

![Hierarchical decomposition and mapping](image)

Figure 3-2. Hierarchical decomposition and mapping
One-to-one mapping does not mean to imply that a particular FR can only have one associated DP. Rather, it says that an FR can only have one associated DP in any one particular system hierarchy.

Hierarchical decomposition allows the designer to focus on one level of detail at a time. Rather than be concerned with the more intricate aspects of a design from the start, the designer can focus on making sure the design is functionally correct before dealing with the details of implementation. This approach ensures that the customer needs are satisfied. Decomposition is also necessary to distinguish functional coupling caused by parent-child relationships, which is intrinsic and unavoidable, and coupling caused by the selection of design parameters, which is relational and avoidable.

3.5 Zigzagging

Since decisions at higher levels in the hierarchy affect the view of the problem at lower levels, the designer goes through a process of zigzagging between the domains to decompose the design problem. For example, before a given level of functional requirements can be decomposed, the corresponding design parameters for those requirements must be chosen. Once all of the DPs have been chosen, the FRs can be decomposed into sub-requirements. The process is then repeated.

This method of traversing the design domains in decomposition is what has been termed zigzagging. Zigzagging is a constraint in the design process that should always be followed.
The designer follows this approach until he has decomposed the problem to such a point that the solution to the remaining sub-problems is known [6]. Zigzagging is illustrated in figure 3-3.

3.6 Design Matrix

Once all of the design elements have been specified at a particular level in the hierarchy, the relationships among the various elements is defined in a design matrix. The design matrix relates two vectors in two different design domains to each other, either \{FR\} to \{DP\} or \{DP\} to \{PV\}. Mathematically, the relationships can be written as:

\[
\{FRs\} = [A] \cdot \{DPs\} \text{ for product design}
\]

\[
\{DPs\} = [B] \cdot \{PVs\} \text{ for process design}
\]
where \([A]\) and \([B]\) are the design matrices. The elements of the design matrix are
determined from the following set of equations [12]:

\[
\Delta F_{R1} = \left( \frac{\partial F_{R1}}{\partial D_{P1}} \right) \Delta D_{P1} + \ldots + \left( \frac{\partial F_{R1}}{\partial D_{Pn}} \right) \Delta D_{Pn}
\]

\[
\ldots
\]

\[
\Delta F_{Rn} = \left( \frac{\partial F_{Rn}}{\partial D_{P1}} \right) \Delta D_{P1} + \ldots + \left( \frac{\partial F_{Rn}}{\partial D_{Pn}} \right) \Delta D_{Pn}
\]

where each element in the design matrix is defined as:

\[ A_{ij} = \frac{\partial F_{Ri}}{\partial D_{Pj}} \]

Once a tolerance is specified for the matrix element values, the elements with values over
the tolerance can be represented with an “X” while those elements with values below get
an “O”. This simplification allows the designer to concentrate on the qualitative meaning
of the design matrix instead of the quantitative one. For effects that can not be described
mathematically, the “X” represents a significant effect while the “O” represents a
negligible effect.

To satisfy the Independence Axiom, the design matrix must be either diagonal or
triangular. When the design matrix is diagonal, each FR can be controlled independently
of the other FRs by its associated DP. This is called an uncoupled design. When the
design matrix is triangular, the independence of the FRs can be maintained by imposing
an order on the decomposition and change of the DPs. This is called a decoupled design.
Otherwise, the design matrix is considered \textit{coupled} and independence of FRs is impossible. A coupled design is generally a bad design [13].

\[
\begin{align*}
\{\text{FR}\} &= \begin{bmatrix} X & O & X \\ O & X & X \end{bmatrix} \{\text{DP}\} \\
\{\text{FR}\} &= \begin{bmatrix} X & O \\ X & X \end{bmatrix} \{\text{DP}\} \\
\{\text{FR}\} &= \begin{bmatrix} X & X \\ X & X \end{bmatrix} \{\text{DP}\}
\end{align*}
\]

(a) Uncoupled \quad (b) Decoupled \quad (c) Coupled

\textbf{Figure 3-4. Examples of the three types of design matrices}

3.7 Independence Axiom

\textit{Axiom 1 Independence Axiom}

Maintain the independence of the functional requirements [12]

The first axiom in axiomatic design is the Independence Axiom. It states that the independence of FRs must be maintained in order to achieve good design. Since FRs are by definition independent, any dependence is introduced during the mapping process between FRs and DPs. In order to satisfy the first axiom, the DPs for a given set of FRs must be chosen such that the intrinsic independence of FRs is maintained. The effect of a DP on an FR is described by an element in the design matrix. Only diagonal or triangular matrices satisfy the Independence Axiom. That is why they are referred to as uncoupled and decoupled designs, respectively.

The significance of the Independence Axiom (and the Information Axiom) lies in the significant improvements realized in the performance, robustness, reliability, and
functionality of those designs that satisfy the design axioms [14]. A design which satisfies the Independence Axiom gains a lot in controllability over coupled designs. Since each DP effects only its referent FR, the DPs can be varied independently until all FRs are satisfied. With a decoupled design, the DPs must be varied in a prescribed order to satisfy the FRs. In a coupled design, the FRs can only be satisfied through a complicated feedback sequence which involves most of the DPs at any given time. Although the FRs can be satisfied in a coupled design, the procedure for doing so is much more complex than for those designs which satisfy the Independence Axiom. This conclusion is supported by many case studies presented by the various authors referenced by this thesis.

3.8 Information Axiom

Axiom 2  Information Axiom

Minimize the information content of the design. [12]

The second design axiom is the Information Axiom. While the Independence Axiom determines the acceptability of a set of possible solutions, the Information Axiom selects the best solution out of the set of acceptable solutions. Axiom 2 states that the “best” solution is the one with the minimum information content. It provides a quantitative means for evaluating design. The information content is defined as the probability a set of DPs will satisfy a set of FRs. Mathematically, the relationship is as follows:
Information \( I = \log_2 (1/p) = \log \left( \frac{\text{system range}}{\text{common range}} \right) \)

The designer obtains the probability \( p \) by examining the graph of the probability distributions of the FR design range and the DP system range for each FR-DP pair. The probability of success for each FR-DP pair is calculated as the area of intersection between the design range and the system range. This area is called the common range. The results are then aggregated to get the overall probability.

![Figure 3-5. Probability distribution functions for FRs and DPs](image)

As the probability \( p \) increases, the information content \( I \) decreases. To satisfy the Information Axiom, the designer should attempt to maximize the common range for the
FRs and DPs in his design. This increases the probabilities and decreases the information content.

3.9 Remarks

Axiomatic design theory provides a framework for designers to follow during the design process. Through diligent application of the design axioms, a designer can evaluate the quality of a proposed design in a rational manner that is supported by easily understood analytical results. The axiomatic design process guides designers at all levels of the design, encouraging them to consider alternatives at all levels and to make explicit choices between those alternatives. The axiomatic design method has been successfully applied to industrial practice to support these claims [6, 11].

The work in thesis focuses on the mapping between functional requirements and design parameters and assesses solutions utilizing the design matrix and the Independence Axiom. The work of the other members of the Axiomatic Design Group deals with the remaining aspects of the theory, such as hierarchical decomposition, zigzagging, and the Information Axiom. Please review [6] for more information on the axiomatic design software effort.
Chapter 4. Thinking Design Machine

This thesis project implements a segment of the Thinking Design Machine concept outlined in a paper by Professor Nam P. Suh of MIT and Shinya Sekimoto of the Toshiba Corporation in Japan. In that paper, the concept of an “intelligent machine” or Thinking Design Machine (TDM) is described. The following chapter presents an overview of the TDM concept. Since the Ideation Database is a portion of the TDM concept, it is useful to review the entire system to see where the database fits in. By examining the TDM, the database design developed in the thesis project can be compared against the database schema proposed by Suh.

4.1 TDM Concept

Although many computer aided design tools have been developed, they generally focus on the modeling and drawing portions of the design process. The focus of the TDM is on the early phase of the process, the conceptual design phase. This is where the most critical design decisions are made and where computer automation could yield the greatest benefits in terms of robustness, reliability, and overall quality of the product.
The goal of the TDM is not to assess the early efforts of human designers but to generate creative designs or design concepts which are superior to those currently possible. It uses the axiomatic design method to define, create, and assess design solutions. Based on the AD principles and focusing on the FR and DP domains specifically, the TDM performs the following four steps:

Step 1. Definition of functional requirements (FRs)
Step 2. Ideation or creation of ideas (DPs)
Step 3. Analysis of the proposed solutions to choose the best solution
Step 4. Checking the final solution

In the first step, a preliminary set of functional requirements is defined. In Step 2, the TDM database is consulted to create a collection of sets of DPs satisfying the FRs. The
proposed solutions are then analyzed in Step 3 against the design axioms and other physical constraints. Based on that analysis, the TDM selects the best solution. In Step 4, the TDM checks the final solution against the original customer needs.

4.2 Architecture of the TDM

The architecture of the TDM consists of four modules. Each module corresponds to a step in the TDM concept. Since Step 1, the definition of FRs, is subjective, the software module governing it acts mainly as a check on the work of the designer. It verifies that the requirements are being correctly stated in the functional domain and that they are being defined in a solution-neutral environment. Unfortunately, it is impossible for the system to tell that these constraints are being filled. However, the software can prompt and step the user through the process and alert him to the constraints at the appropriate times. By interacting with the designer in such a manner, the system can attempt to ensure that the constraints are followed.

An ideation software module handles Step 2. By consulting the database, the system finds a plausible set of DPs for each FR being considered. For example, if the FR is “generate power”, the system may find some DPs such as “motor”, “solar panel”, or “windmill”. The designer may also propose new DPs for the FRs during this step. The system then creates the design matrices relating the FRs to the plausible DPs.

An analysis software module has the task of determining whether the designs generated by the ideation software are acceptable. If the design matrix is diagonal, the analysis software concludes that it is an acceptable design. If the matrix is triangular, the
software recommends an ordering on the design parameters that will maintain independence of the FRs. If the matrix is neither, then the design might be coupled. The analysis software attempts to reorder the matrix to yield either a diagonal or triangular matrix. If reordering yields neither case, the analysis software identifies possible regions in the matrix for the designer to re-examine.

The last module in the TDM is the system software module. Its job is to verify that the final solution satisfies the original customer needs and to arrange the selected physical components into a working physical system. The sequence of control for the components is dictated by the design matrix. At this point, the TDM would iterate through the process again if it was necessary.

4.3 Database for the TDM

The database is a crucial portion of the TDM. It needs to contain sufficient knowledge to allow it to assist designers by communicating possible solutions. The database should also contain the design knowledge of many designers to give it the largest possible search space with which to synthesize solutions.

In the TDM the FRs and DPs are described and stored as text strings. Once an FR is given, the TDM analyzes the meaning of the FR and searches the database for the corresponding DPs. The search is made through parsing and keyword matching. The database also contains a set of technical terms and their corresponding synonyms, antonyms, etc.
The knowledge representation of the TDM database is expressed in PROLOG as two types of functions:

\[ \text{fr-dp("FR", ["DP1", "DP2", ...])} \]

and

\[ \text{dp-fr("DP", ["FR1", "FR2", ...])} \]

The functions specify the nature of the relationships between the FRs and DPs. For example, the fr-dp function states that functional requirement "FR" is satisfied by one of the design parameters "DP1", "DP2", etc. The TDM database is essentially a collection of these functions. Although they are interchangeable, the presence of both types of functions in the database increases the probability of finding a new solution [15].

4.4 Remarks

The design and development of the TDM is a long-range goal of the software efforts in the Axiomatic Design Group. To that end, the thesis project focuses on realizing Step 2 of the TDM. The general framework outlined by Suh for the ideation software of the TDM is applicable to the work of the thesis. However, the project follows a different approach in developing the knowledge representation for the database. Rather than incorporate the representation presented in the TDM paper, this thesis develops a new knowledge representation. Chapter 5 describes the new representation.
Chapter 5. Ideation Database Knowledge Representation

The importance of the knowledge representation to the success of the Ideation Database was recognized early on in the course of the thesis project. Since there was very little consensus in the design community regarding the language of design, the representation needed to deal with the ambiguity before any meaningful work towards systematizing the design process could proceed. In the absence of any particular standards, the development of the knowledge representation focused on supporting the functions of the Ideation Database.

Given a set of functional requirements, the Ideation Database should be able to search its elements for an appropriate set of design parameters to satisfy those requirements. The suggested design parameters should then be analyzed to determine the structure of the corresponding design matrix. This database system should be able to make this determination as efficiently as possible. Additionally, the database needs be extensible, and maintenance overhead should be kept to a minimum.

In order to satisfy these requirements, an effective knowledge representation must be developed to support the database functions. A well-designed representation reduces the overall complexity of the analysis routines in the software at the cost of a little more maintenance overhead for the database. This chapter presents the knowledge
representation developed during the course of the thesis work to support the functions of the Ideation Database system.

5.1 Object Relationships

In order to accommodate the requirements of the database, the knowledge representation is constructed using object-oriented methods and consists of two types of data objects: FR objects and DP objects. The FR objects are the primary objects in the database, and each FR object is linked to multiple DP objects. The DP objects only have meaning with respect to the FR objects. This means that although there may be FR objects without links to DP objects, there are no DP objects without links to FR objects. Additionally, DP objects are linked only to one FR object although they can effect many other FR objects. This behavior is equivalent to implementing only the fr-dp function types in the TDM. Figure 5-1 presents a graphical example of these relationships.

![Figure 5-1. Relationships between FR and DP objects](image-url)
Figure 5-1 illustrates a sample set of objects and their relationships as they would appear in the database. FR1 has a link to DP1, FR2 has links to DP2a, DP2b, and DP2c, and FR3 has no direct links to DPs. However, FR3 does have indirect links to both DP1 and DP2b. These indirect links specify that those DPs effect FR3 even though no direct link between them exists. A direct link between an FR and DP means that the DP satisfies the FR. An indirect link between an FR and DP means that the DP affects the FR. Direct links are defined when a DP is associated with an FR. Indirect links are made at runtime. The way those links are determined is described in section 5.3.

5.2 Object Data Structure

The data structure of the objects can be viewed as an extension to the standard text string method of describing FRs and DPs. Table 5-1 presents the data structure for each type of object.

<table>
<thead>
<tr>
<th>Functional Requirement</th>
<th>Design Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>ID</td>
</tr>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Initial State</td>
<td>State Transform</td>
</tr>
<tr>
<td>Final State</td>
<td>FR ID</td>
</tr>
<tr>
<td>Attributes</td>
<td>Attributes</td>
</tr>
<tr>
<td>Notes</td>
<td>Notes</td>
</tr>
</tbody>
</table>

Table 5-1. Data fields for FR and DP objects

Both objects share a set of common fields. The ID field is a unique numerical identifier assigned and used by the database to identify the object in storage. The Name
field is a text string used to describe the intended behavior of the object. It serves the same purpose as the text strings associated with other FR or DP descriptions. The Notes field is used to store any annotations concerning the object a designer may have.

Unlike the other fields, the Attributes field is really not a field at all. It is a logical bin used to accommodate whatever generic attributes a designer may want to add to an object. Each attribute has fields for a name, value, and type as well as fields for an owner ID and owner type. The name, value, and type fields give the designer the flexibility to define attributes specific to the object. The presence of such attributes further refines the definition of an object. The owner ID and owner type fields identify the object containing this attribute.

What remains are a group of fields specific to the respective objects. The FR ID field in a DP object contains the unique ID of the FR that initially defined the DP. The Initial State and Final State fields of an FR object describe the state of the environment before and after the completion of the function represented by the FR. They may be empty to signify an environment in rest or no effect. The State Transform field of the DP object defines how the DP satisfies the FR and maps to the FR object’s Initial and Final State fields. It contains a keyword for each effect the DP object has on the environment.

<table>
<thead>
<tr>
<th>Functional Requirement</th>
<th>Initial State</th>
<th>Final State</th>
<th>Design Parameter</th>
<th>State Transform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate power</td>
<td>power</td>
<td></td>
<td>Motor</td>
<td>power</td>
</tr>
<tr>
<td>Increase flow</td>
<td>flow</td>
<td>flow + delta</td>
<td>Vertical faucet control</td>
<td>flow</td>
</tr>
<tr>
<td>Control hot water flow</td>
<td>flow</td>
<td>flow + delta</td>
<td>Hot water knob</td>
<td>flow temperature</td>
</tr>
<tr>
<td>Stop water flow</td>
<td>flow</td>
<td></td>
<td>Stopper</td>
<td>flow</td>
</tr>
</tbody>
</table>

Table 5-2. Example of state field values
Table 5-2 presents some examples of the state relationships between FRs and DPs.

The descriptions of the fields above serve only as a guideline to their use. The software has no hard-coded requirements on the content of the fields. However, the function of certain fields dictates the format required. For efficiency and correctness, the state fields should only contain keywords. Articles and other extraneous text should be eliminated. None of the other fields in the objects need to adhere to this provision. The reason for the provision is explained in the next section.

5.3 Interaction Model between FRs and DPs

One feature of this knowledge representation which distinguishes it from the standard representation for FRs and DPs is the way effects between those objects are determined. The effects are used to fill in the design matrix for each set of FRs and DPs. In the database of the TDM proposed by Suh, the effects are represented by functions in the database. For example, the function

\[
dp-fr\left(\text{"gear"}, \left[\text{"transmit rotary motion"}, \text{"change torque"}, \text{"change revolution"}\right]\right)
\]

defines a relationship between each of the FRs in the list and the DP “gear”. If the TDM needed to determine whether a certain FR was affected by the DP “gear”, it would check for the presence of the FR in this dp-fr function.

The main difficulty with using the database of functions as presented by Suh in the TDM paper is in updating the records. In order to be sure that all possible effects are
present in the database, the designer entering a new FR would have to manually traverse
the entire database of dp-fr functions, entering the FR into a function when there is an
appropriate effect. As the database gets larger, this becomes more and more unrealistic.

As another way to update the database, the designer could elect to make
modifications only when a design matrix is presented. The designer would examine the
elements of the matrix and make modifications to the database when an element is
incorrect. The problem with this method is that, while it is evident that effects are effects
(an “X” in the matrix corresponds to a distinct relationship in the database), it is unclear
what the situation is when an element in the design matrix is a non-effect (it is ambiguous
what the “O” means). That situation is generated in two ways: there really is no effect or
the effect has yet to be entered into the database. If it is the second case, the designer
simply updates the database and proceeds. However, if the designer determines that it is
the first case, there is no easy way to capture the fact that there is no effect in the existing
framework. That means that all future designers become burdened with the task of
determining that non-relationship as well.

The extension proposed to handle this problem is the use of the state fields
described above. When the Ideation Database needs to determine whether there is an
effect between a particular FR and DP, it compares the values of the initial and final state
fields of the FR against the state transform of the DP. The comparison is made by
looking for a match between the keywords of the state transform and the keywords of the
initial and final states. This is the reason behind the previously mentioned provision
against the use of extraneous words in the state fields. Figure 5-2 illustrates this process
graphically.
The system uses the results of the interaction model to fill in elements of the design matrix. When the Ideation Database examines design matrix element $A_{ij}$, it compares the contents of the initial and final state fields of $F_R_i$ with the state transform field of $D_P_j$. If there is a match, the design matrix element $A_{ij}$ receives an “X” to signify the effect. Otherwise, the element receives an “O”.

This knowledge representation simplifies the update process. When a new FR is entered, all the designer needs to do is enter the appropriate initial state and final state. The designer no longer has to update all of the affected DPs when entering a new FR. As long as the state fields are defined correctly, the Ideation Database can determine when the FRs and DPs affect each other at runtime by following the interaction model above.
The update process becomes an order(1) routine, an improvement over the order(n) of the TDM function database.

5.4 Remarks

Originally the addition of extra fields into the object data structure was an attempt to provide the Ideation Database system with more keys to search on. By allowing a designer to specify several different attributes for each object, the system reduced the reliance on the description string. During the course of the thesis work, the fact that every designer had his own way of describing things became apparent. Rather than attempt to develop a standard syntax for the description of FRs and DPs, more fields were added to the knowledge representation. This gave the designer more flexibility in defining an FR or DP and gave the software more fields to search for possible matches. However, as the knowledge representation developed, the state elements evolved into key elements in the analysis of effects between FRs and DPs. As mentioned before, the problem with maintaining a list of associated elements for each FR and DP was in updating the records. As discussed above, using the state fields to determine effects at runtime saved a significant amount in system overhead.
Chapter 6. Ideation Database System Architecture

With the knowledge representation as the backbone, this thesis project proceeds to implement the Ideation Database system. This chapter analyzes the design of the system using the axiomatic design method. First, the customer needs are determined. Then, the functional requirements are defined, and the design parameters are chosen. Two level of decomposition are presented.

6.1 Top Level FRs and DPs

According to the axiomatic design method, the first task in the design of the system is the determination of the customer needs. At the highest level, the customer need for the Ideation Database system are:

CN. Assistance for the designer during design process

The customer need specify the attribute desired in the software. To be effective, the Ideation Database must satisfy this need. A design which provides many useful features yet does not satisfy the original customer needs is a bad design.
Given the customer need, the following root FR and DP are defined

FR. Assist designer manage design objects

DP. Ideation Database system

From the root, the following top-level functional requirements are defined:

FR1. Store and retrieve information

FR2. Interact with the designer

The FRs are essentially restatements of the customer needs in functional terms. They describe the functions that the software must provide in order to satisfy the customer need.

To satisfy these requirements, the following design parameters are chosen:

DP1. Database

DP2. User Interface

In software, the design parameters represent the modules that are used to satisfy the functional requirements. For the Ideation Database, a database module handles storage and retrieval of design elements, and the user interface module handles presenting information to the designer.

Once the FRs and DPs are chosen for a particular level of the hierarchy, the design matrix can be created. The highest-level FRs and DPs yield the following matrix:
The matrix is decoupled. Although the user interface is primarily used to present information to the user, it effects the storage and retrieval of information since it contains the control elements (e.g., the forms and callbacks for data entry) needed to access the database. Since this is a decoupled design, the design parameters must be decomposed in a particular order to maintain independence for the functional requirements. For the Ideation Database, FR1 “Store and retrieve information” should be decomposed first. This fixes the effect of DP1 “Database”. Once that effect is determined, FR2 “Interact with the designer” can be decomposed.

6.2 Decomposition of FR1 “Store and retrieve information”

The FR “Store and retrieve information” can be decomposed as follows:

FR1.1 Handle basic storage and retrieval
FR1.2 Handle queries for information

These FRs are chosen given the DP “Database”. If another DP had been chosen to satisfy FR1, the decomposition would be different.

The DPs for these FRs are:
The two DPs are both specific to a database. Rather than considering these sub-DPs as just the decomposition of the parent DP “Database”, they are the reflection of the choice of FR as well. An FR alone or a DP alone does not determine the decomposition. It is the two together that determines the elements of the next level.

At this level, the design matrix is:

\[
\{\text{FR}\} = \begin{bmatrix} X & O \\ O & X \end{bmatrix} \{\text{DP}\}
\]

This is an uncoupled design. The work of DDE does not affect how SQL interacts with the database. More information about DDE and SQL can be found in [10]. Decomposition stops here since DP1.1 and DP1.2 are both leaves.

6.2 Decomposition of FR2 “Interact with the designer”

Since the design matrix at the previous level was coupled, the value of DP1 “Database” had to be determined before FR2 could be decomposed. With FR1 decomposed and DP1 determined, FR2 can be decomposed with the structure of the database as a constant.

The FR “Interact with the designer” can be decomposed as follows:
FR2.1 Present database information to the designer
FR2.2 Assist designer in analyzing requirements

In addition to being affected by the choice of FR2 and DP2, the two functional requirements at this level are also affected by the result of the decomposition of FR1 and DP1. The coupled nature of the top level design matrix is evidence to this fact. Since FR1 and DP1 were decomposed first, the two sub-FRs at this level can be defined taking the result as a given.

The DPs for these FRs are:

- DP2.1 Database View
- DP2.2 Analysis View

These two DPs define the two main modules of the user interface. Unlike the two sub-DPs for DP1 which are leaf nodes, the implementation of these two sub-DPs is unclear at this point. Further decomposition (which will not be shown here) is necessary.

At this level, the design matrix is:

\[
\{\text{FR}\} = \begin{bmatrix} X & O & X \\ O & X \end{bmatrix} \{\text{DP}\}
\]

The decomposition of DP2 is also uncoupled. Therefore, the design and decomposition of both modules of the user interface can continue independently and in parallel.
6.3 System Hierarchy

The decomposition above results in the following system hierarchy in the FR and DP domains:

![System hierarchy for the Ideation Database](image)

Figure 6-1. System hierarchy for the Ideation Database

The ghost boxes represent further decomposition not presented here.

6.4 Remarks

By analyzing the architecture of the Ideation Database using the axiomatic approach, two things are gained. The validity of the axiomatic approach in software design is demonstrated and the decision-making process governing the design of the system is documented.

Axiomatic design provides a rational framework to a software design process traditionally dominated by trial-and-error methods. It also provides a better design
documentation structure than other software design methods such as Object Modeling Technique (OMT) and IDEF0 which focus mainly on program flow and structure. Following the axiomatic approach, the designer can determine the rationale for a particular design as well as the recommended structure of data flows [14].
Chapter 7. Ideation Database User Interface

The user interface of the Ideation Database consists of two modules. The primary module is the Analysis View. The Analysis View allows the designer to enter a set of FRs into a form. It then uses the FR set to generate a collection of sets of appropriate DPs based on the information contained in the database. It also displays a design matrix for each set of DPs. The secondary module is the Database View. The designer uses the Database View to view the elements of the database, update existing elements, enter new elements, or delete invalid ones. The designer also has access to a multiple field keyword search which sort the records according to the results of the search.

![Application Structure Diagram](image)

**Figure 7-1. Ideation database application structure**

The application structure for the Ideation Database is a standard document-view form with extensions into a database. The Analysis View and Database View are both
views into the same document. The document initializes and owns the link to the database which the views access through public member functions. The views and the document are owned and initialized by the application. The structure is presented graphically in figure 7-1.

7.1 Analysis View

The Analysis View is a worksheet the designer uses to evaluate a set of functional requirements at a given level in the hierarchy. The interface is presented in figure 7-2.

![Figure 7-2. Analysis View form](image-url)
The designer begins by entering a string describing the FR he wishes to use into the Functional Requirements edit box (A). This brings up a dialog listing the functional requirements currently found in the database whose names most closely resemble the string entered. For convenience, the search string is displayed in the top edit box on the dialog. The dialog is shown in figure 7-3.

If the designer finds an appropriate FR, he selects it and presses the “OK” button. The text of the FR is then copied into the FR list on the Analysis View form (B). The designer repeats this process until all the desired FRs are entered into the FR list. If at any point, an appropriate FR was not found, the designer should enter the Database View and add the FR and an associated DP. Also, if the designer makes a mistake in selecting a particular FR, the designer can delete the FR by selecting it in the FR list and pressing the “Delete FR” button.

![Figure 7-3. FR selection dialog](image)

---

63
Once all of the desired FRs are entered, the designer hits the “Analyze” button. The Ideation Database takes the FR list and translates it into a list of FR IDs corresponding to objects in the database. The system then fills in a data structure with all the possible combinations of DPs for this set of FRs. The combinations are generated using a recursive algorithm.

When all the possible combinations of DPs have been generated and stored by the system, it loads the first DP combination into the DP list (D) and calls a function to load the Matrix list (C). To load the Matrix list, the Ideation Database analyzes the set of FRs against the current set of DPs and applies the FR-DP interaction model to each combination of FR and DP to determine if an effect exists. When all of the design matrix elements have been determined, the matrix is displayed. The designer then uses the “Prev DM” and “Next DM” buttons to scroll through the DP combinations. With each new combination the designer requests by scrolling, the Ideation Database generates a new design matrix for the set of FRs and DPs. The design matrices are not stored in memory. When the designer is done with this level in the hierarchy, the “Clear FRs” button resets all of the lists.

7.2 Database View

The designer uses the Database View for interactions with the database. This includes adding, updating, and deleting records. It is also through the Database View that designers enter and edit DPs. The interface is presented in figure 7-4.
By using the "Record" menu and buttons on the toolbar (not shown), the designer can navigate through all of the records contained in the database. The designer edits a record by modifying the contents of the field and moving to another record. The changes are automatically sent to the database. Adding a record presents the designer with a blank form to fill in. As with editing, the changes are automatically sent to the database once the designer moves to another record. To delete the current FR, the designer would use the "Delete" option under the "Record" menu.

In order to view the DPs associated with a particular FR, the designer presses the "View DPs" button on the FR form. This brings up the DP dialog shown in figure 7-5.

Figure 7-4. Database View form
Editing works the same way as with FRs. However, when making modifications to the DP, no changes are committed until the designer hits the “OK” button on the dialog. The designer can scroll through the possible DPs for this FR by pressing “Previous” and “Next” on the dialog. To delete the DP, the designer presses “Delete DP”. When the “Add DP” button is pressed on the FR form, the DP dialog is shown with an empty form. The designer fills in the fields with the appropriate values and presses “OK” to commit the addition to the database.

Both the FR and DP objects have attributes. As described in the knowledge representation, the attributes are generic holders which have a name, a value, and a type.
To add an attribute to an object (FR or DP), the designer presses the “Add Attribute” button on the appropriate form. This brings up the attribute dialog in figure 7-6.

The designer fills in the fields with the desired information and presses “OK”. The only requirement on the fields is that there is no whitespace in the “Name” field since that field is used in the attribute management functions.

To edit an attribute, the designer selects the attribute in the attribute list and presses “Edit Attribute”. This bring up the attribute dialog with the fields filled with the data of the selected attribute. The designer makes the desired modifications and presses “OK”. To delete an attribute, the designer selects the attribute in the attribute list and presses “Delete Attribute”.

In addition to being able to view, edit, and add objects to the database, the Database View provides a search feature to assist the designer in finding FRs. The search dialog is presented in figure 7-7.
The designer enters the appropriate keywords for each field in the FR object. The “Search Threshold” is the maximum number of matches that an FR can achieve yet still be discarded. Any of the fields may be empty.

When the designer presses the “Search” button, the Ideation Database examines every FR object in the database and ranks them according to the number of words matched in each field. When it is done ranking the objects, the system sorts the records according to rank and filters out those that do not rank higher than the search threshold. The results are shown in the Database View. It looks the same as before, except now the records are sorted and filtered according to the search criteria. To refresh the Database View to include all of the records again, the designer uses the “Show All Records” option in the “Record” menu.
7.3 Remarks

There are still elements of the user interface and the user interaction process that should be improved to link the Analysis View with the Database View. For example, if the user enters a FR string in the Analysis View that has no matches in the database, the system should present the appropriate form and prompt him to enter a new FR. As the system stands, the two are independent views of the same data that do not interact. It is up to the user to move between the two.

Aside from that note, the Ideation Database system behaves very much as it should. If the designers have maintained the database objects correctly, the Ideation Database will be able to synthesize the information and present the user with all the possible sets of DPs and their design matrices given the set of FR entered.
Chapter 8. Implementation

This chapter discusses some of the implementation details encountered during the course of development. It is not meant to be a comprehensive view of all of the implementation details in the Ideation Database, but a place where some of the issues specific to this project are discussed.

8.1 Link to a Database System

Rather than create a database system from scratch, the Ideation Database maintains a link into a Microsoft Access database for storage and retrieval of elements and for sorting and filtering sets of elements. The communication between the Ideation Database system and the Access database is handled by a Microsoft extension to OLE called Microsoft Foundation Class (MFC) Data Access Objects (DAO). The DAO classes in MFC support opening several different types of databases [10]. Microsoft Access was chosen for its widespread use and availability. By using an existing database management system (i.e., Access), the thesis work could concentrate on developing the knowledge representation and analyzing the relationship between FRs and DPs.
8.2 Managing Relationships in Access

Access has no explicit relationship mechanism. Although the user can define relationships in the schema, those relationships did not create any linkages in the representation of the objects that were not already present. The effect seemed purely cosmetic.

In order to relate a DP to an FR and attributes to either, the object IDs are used to create symbolic links. The links behave like pointers to objects. The DP objects have a field, FR ID, which corresponds to the unique ID of the corresponding FR object. Attribute objects have two fields, Owner ID and Owner Type, which when used together select the owning object. For example, the DPs for a given FR object are obtained by filtering the database elements to find those DPs whose FR ID field matches the ID of the FR. Figure 8-1 presents an example.

![Figure 8-1. Implicit relationships in Microsoft Access](image)

In this example, DP1 and DP3 are related to FR1 by virtue of the FR ID field. The FR ID field value for each of those DPs matches the unique object ID of FR1. This creates
the implicit relationship mentioned above. DP2 is not related to FR1 since the value in
its FR ID field, 2, does not match the ID of FR1.

8.3 Generating Combinations of DPs

Once the designer enters a set of functional requirements to be analyzed, it is the
Ideation Database system’s task to search for all the possible DPs and arrange the results
into a collection of sets of DPs. The difficulty here is that nothing is known a priori about
the number of FRs to be analyzed or the number of DPs associated with each one. These
ambiguities lead many conventional combinatorial methods to be unless.

In order to gain some insight on the problem, consider an example. Say there are
three FRs, FR1, FR2, and FR3, and each of those FRs has two DPs, DPia, and DPib,
where i is 1, 2, or 3. All the possible combinations of DPs given this information can be
listed and ordered as follows (assuming the order of FR1, FR2, FR3):

1 : DP1a, DP2a, DP3a 5 : DP1b, DP2a, DP3a
2 : DP1a, DP2a, DP3b 6 : DP1b, DP2a, DP3b
3 : DP1a, DP2b, DP3a 7 : DP1b, DP2b, DP3a
4 : DP1a, DP2b, DP3b 8 : DP1b, DP2b, DP3b

Upon closer examination of the sequence, a pattern emerges. The highest order elements
(the DPs for FR3 in this case) change with every set. The next highest order elements
(the DPs for FR2) change with every two sets. The lowest order elements (the DPs for
FR1) change with every four sets. In terms of the Ideation Database, the highest order
FR is the last FR in the FR list on the Analysis View form, and the next highest order FR is the next to last FR in the list and so on.

If the result is generalized (which can be done with the benefit of more examples which will not be included here), the following three rules emerge:

1. The highest order elements change with every iteration.
2. At any particular level below the highest, the elements change with every n iterations where n is the length of a complete cycle of distinct elements in the level above. This means that each element is repeated n times in its place in the cycle.
3. A cycle is defined as a sequence of the m DPs associated with the FR for that level, with each DP repeated n times.

Given those rules, the software uses a recursive approach for generating all of the possible combinations. The DP combinations are stored in a two-dimensional matrix, where each row in the matrix is a different combination of DPs, and each column corresponds to one of the FRs in the FR list. In the data structure, column 0 corresponds to the first FR in the FR list.

The recursive algorithm works by first filling in the current row with one complete cycle of the DPs corresponding to the FR at that level. Each element is repeated n times, where n is the length of the previous cycle. Since the completed cycle at this level will be k times longer than the length of the previous levels, where k is the number of DPs at this level, the elements of those previous levels need to be duplicated to fill out the appropriate length in the matrix. For each level, (k - 1)*n elements need to be copied at the previous level (the level above). Once those procedures are completed, the
recursive call is implemented with the number of FRs, the length of a cycle in the current level, and the index of the next level as parameters. The first time the recursive function is called, its parameters are the number of FRs, the index of the highest level, and 1 (the number of times the first set of DPs needs to repeat in a cycle). The function returns when the index of the current level is -1.

As an example, suppose the designer chooses the following three FRs:

FR A: [ DP A1, A2 ]
FR B: [ DP B1, B2, B3 ]
FR C: [ DP C1, C2 ]

To generate the combinations of DPs, the Ideation Database calls the recursive algorithm.

The initial parameters are 3 (the number FRs), 2 (the index of the highest level), and 1.

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4</th>
<th>Step 5</th>
<th>Step 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. -1,-1,C1</td>
<td>1. -1,B1,C1</td>
<td>1. A1,B1,C1</td>
<td>1. A1,B1,C1</td>
<td>1. A1,B1,C1</td>
<td></td>
</tr>
<tr>
<td>2. -1,-1,C2</td>
<td>2. -1,B1,C2</td>
<td>2. A1,B1,C2</td>
<td>2. A1,B1,C2</td>
<td>2. A1,B1,C2</td>
<td></td>
</tr>
<tr>
<td>5. -1,B3,-1</td>
<td>5. -1,B3,C1</td>
<td>5. A1,B3,C1</td>
<td>5. A1,B3,C1</td>
<td>5. A1,B3,C1</td>
<td></td>
</tr>
<tr>
<td>7. -1,-1,-1</td>
<td>7. -1,-1,-1</td>
<td>7. -1,-1,-1</td>
<td>7. -1,-1,-1</td>
<td>7. -1,-1,-1</td>
<td></td>
</tr>
</tbody>
</table>

Table 8-1. Example of the recursive algorithm
Table 8-1 shows the sequence followed by the recursive algorithm to fill the matrix of DP combinations (the -1’s represent empty elements).

Step 1 is the result of the first call to the recursive function. Steps 2 and 3 result from the second call. Step 4, 5, and 6 result from the third call. The fourth call to the recursive function returns since the index is -1.

8.4 Development Environment

The Ideation Database system was developed in Microsoft Visual C++ 4.0 on a Dell Pentium Pro 180 running Microsoft Windows NT Workstation 4.0. As mentioned before, the database management system used was Microsoft Access. The Visual environment made creating and modifying the user interface fairly straightforward. The inclusion of Microsoft Foundation Class (MFC) libraries greatly simplified the task of linking to the Access database. To run the Ideation Database, a user would just need to have a 32-bit operating system such as Win95 or Win NT, the Ideation Database executable, and the Access file containing the database schema. The user does not need to have Visual C++ or Access.
Chapter 9. Case Study: Lamp Life Improvement Project

This chapter presents a case study of a lamp life improvement project undertaken at a semiconductor process equipment company. The intention is to demonstrate how the knowledge representation and Ideation Database would be used in industry. It is generated from information gathered by the author from on-site visits and electronic documents. Since this case study primarily serves an illustrative purpose, only a few levels in the system hierarchy are presented here.

9.1 Lithography Process

In the lithography process, ultraviolet radiation is used to expose an image onto an oxidized layer of a silicon wafer. The wafer is first treated with an ultraviolet (UV) light-sensitive photoresist (PR) film. This involves coating the wafer with the PR and then baking off the solvent and hardening the PR layer. Once the PR layer has hardened, a mask containing the desired pattern is placed over the wafer, and it is exposed to UV light. The radiation that passes through the mask reacts with the PR. The wafer is then developed removing the UV-exposed PR. This leaves the mask pattern on the PR layer of the wafer.
The wafer is baked again to harden the remaining PR in preparation for the etching process. In etching, the exposed portions of the oxide layer are removed from the silicon substrate. The PR is then removed in a solvent solution, leaving the pattern on the oxide layer of the wafer. This process is repeated for as many layers as are desired [4].

9.2 Lamp Life Improvement Project

Lithography utilizes ultraviolet radiation to transfer a pattern from a mask to an oxide layer on the wafer. The UV radiation is generated by filtering and focusing visible light until the desired optical properties are obtained. The light is generated by either a 2.4kW or 3.5kW lamp connected to a variable power supply. The entire illumination subsystem is controlled by an embedded system in the lithography machine which has access to the controls of the variable power supply as well as an array of analog readings on the intensity of the light at various locations on the wafer plane.

At the beginning of lamp life, filters must be used to reduce the intensity of light at the wafer plane to match the maximum scan speed obtainable by the machine. That relationship is governed by the following equation:

\[
\text{Scan Speed (cm/s) \times Dose (mJ/cm^2) = Power (mW/cm)}
\]

The scan speed is the speed at which the pattern on the mask is scanned onto the wafer plane. The dose is a user controlled value specifying the desired intensity of light. The
required dose is controlled by the type of mask pattern being scanned. Its value is obtained in a product specification file. Power refers to the power output of the lamp.

The goal is to adjust the power value to obtain and maintain the maximum scan speed. The scan speed controls throughput, and throughput is the critical performance measure for the machine. In the past, this was achieved by using a set of filters in the light path to reduce the effective power seen at the wafer plane. As the lamp aged, its power output dropped and fewer filters were needed. This proceeded until the power of the lamp without filters fell below a useful threshold.

What the lamp life improvement project proposes is another way to achieve the reduction in lamp output at the beginning of lamp life. Rather than filter away unwanted light, the plan is to reduce the power to the lamp until the desired output value is reached. As the lamp ages, the power value is increased to maintain output until a maximum is reached. At that point, the scan speed is reduced to match the decrease in power. This process continues until the output falls below a useful level.

Running at lower powers preserves the lamp cathode tip. It is the state of the cathode tip that determines the effective output of the lamp.

![Figure 9-1. Cathode tip erosion](image)

(a) new cathode tip  (b) old cathode tip
Figure 9-1 shows a new cathode tip and an old cathode tip. As the lamp ages, the cathode tip erodes, and the useful region of the tip (shown by the gray) diminishes. Running at lower powers at the beginning of lamp life preserves the cathode tip. This prolongs the effective life of the lamp, allowing more wafers to be processed. The rest of this chapter analyzes the illumination subsystem proposed by the lamp life improvement project using the axiomatic design method and the Ideation Database knowledge representation. Only the first few levels of the decomposition are presented for the sake of brevity.

9.3 Development of the Top Level FR and DP

At the highest level the customer need can be stated as:

CN. Increase wafer throughput

That need motivates any improvement made to the lithography machine. It certainly motivates the lamp life improvement project. The project increases wafer throughput by increasing the effective life of the lamp. This allows the lithography machine to run at the maximum step speed for a longer duration, resulting in more wafers processed.

The top level FR and DP for the lamp life project are:

FR. Adjust power to lamp

DP. Power adjustment subsystem
Defined in terms of the Ideation Database knowledge representation, the fields for the top level objects are filled as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Initial State</th>
<th>Final State</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjust power to lamp</td>
<td>lamp-power</td>
<td>lamp-power + delta</td>
<td>max power = 3.8 kW</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>max current = 100 A</td>
</tr>
</tbody>
</table>

Table 9-1. Top level FR for lamp project

<table>
<thead>
<tr>
<th>Name</th>
<th>State Transform</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power adjustment subsystem</td>
<td>lamp-power</td>
<td></td>
</tr>
</tbody>
</table>

Table 9-2. Top level DP for lamp project

The values in the fields further define the objects beyond their description strings. The initial state of one of the environment variables before the FR is some set lamp-power value. After the FR, the final state of that environment variable is its original value plus some set step in power noted as delta. The state transform of the DP is on the lamp-power environment variable. This means that the DP affects the lamp-power variable and, consequently, any FR that deals with lamp-power. This also corresponds to the state fields of the FR which defined it. Although there may be many different variables in the environment, the state fields focus on the variable or variables being affected by this particular FR. There is no one correct way to define the state fields. The values just need to be consistent with how state transforms are defined for the design parameters in the database. The attribute field of the FR contains two constraints on the lamp-power value.
These attributes serve as extra information and do not have an impact on the behavior of the Ideation Database.

9.4 Development of Second Level FRs and DPs

At the next level in the decomposition, the following FRs are defined:

<table>
<thead>
<tr>
<th>Name</th>
<th>Initial State</th>
<th>Final State</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculate constant intensity value</td>
<td>light-intensity</td>
<td>dose = X mJ/cm²</td>
<td></td>
</tr>
<tr>
<td>Measure intensity of light</td>
<td>measurements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adjust power within operating range</td>
<td>lamp-power</td>
<td>lamp-power + delta</td>
<td>max power = 3.8 kW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>max current = 100 A</td>
<td></td>
</tr>
</tbody>
</table>

Table 9-3. Second level FRs for lamp project

Again, the values used in the state fields are open to interpretation. As long as there is consistency between the state fields of the FR and the state transform of the DP, the Ideation Database system can make valid assessments of the effects in the design.

The DPs for the FRs given are:

<table>
<thead>
<tr>
<th>Name</th>
<th>State Transform</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intensity calculation function</td>
<td>light-intensity</td>
<td></td>
</tr>
<tr>
<td>Wafer plane measuring devices</td>
<td>measurements</td>
<td></td>
</tr>
<tr>
<td>Interface to the power supply</td>
<td>lamp-power</td>
<td></td>
</tr>
</tbody>
</table>

Table 9-4. Second level DPs for lamp project
Now that the FRs and DPs for a particular level have been defined, the design matrix can be analyzed.

\[
\begin{bmatrix}
\text{FR1.1 Calculate constant intensity} \\
\text{FR1.2 Measure intensity of light} \\
\text{FR1.3 Adjust power}
\end{bmatrix}
\begin{bmatrix}
X \\
O \\
O
\end{bmatrix}
\cdot
\begin{bmatrix}
\text{DP1.1 Intensity calculation function} \\
\text{DP1.2 Wafer plane measuring devices} \\
\text{DP1.3 Interface to the power supply}
\end{bmatrix}
\]

At this level, the design is uncoupled. That result can be extrapolated from the value of the state fields for the FRs and DPs.

9.5 Using the Ideation Database

With the Ideation Database system, much of the process above is automated. Assuming that the design objects in question are in the database, the designer would determine his functional requirements at a particular level in the hierarchy and enter them into the Analysis View of the Ideation Database. The system would then search for the appropriate design parameters and return the possibilities to the designer along with a design matrix for each possibility. The designer then selects a combination of DPs to use and proceeds with the rest of the design. If an appropriate set of DPs is not found, the designer should enter the Database View and update the database with the appropriate elements. Also if any of the design matrix elements seem incorrect, the designer should consult the database through the Database View and make corrections to the state fields in question.
9.6 Remarks

This chapter makes a simple analysis of the first few levels of decomposition of the lamp life improvement design. The purpose is to illustrate the use of the Ideation Database knowledge representation outlined in chapter 5 with an example from industry. With the amount of flexibility afforded to designers by the knowledge representation, the design objects defined in this chapter only serve to highlight the important points of determining the field values. The point to be most aware of regards the use of the state fields. It really does not matter what value is used in the fields as long as the values in DP state transforms map to those in the FR initial states and final states. The rest of the fields in the knowledge representation serve to give the designer more leeway in describing the design objects.
Chapter 10. Conclusions

The development and implementation of the Intelligent Ideation Database system greatly advances the concept of the Thinking Design Machine. The thesis project and the corresponding implementation clarifies and refines the role of the database within the design process. Also, by determining the type and amount of information needed for the database to perform correctly and efficiently, insight is garnered into the information requirements of the design process.

10.1 Knowledge Representation

An effective knowledge representation must support at least three functions. The primary function of the representation is to support the storage and retrieval of data associated with the design objects (FRs and DPs). The second function is to aid search within the database. The third function of the representation is to provide a framework from which the system can make decisions to aid the designer.

The knowledge representation used by the Ideation Database has elements to support all of those functions mentioned above. The Name and ID fields for each type of object are used to support storage and retrieval. By adding attribute fields and state description fields, the representation gives the Ideation Database more fields on which to
base a search. Rather than employing a binary "found" or "not found" logic, the presence of multiple fields allows the database system to rank each of the records and return those whose rank passes a certain threshold. To aid decision making, the state fields in the representation allow the database system to draw conclusions about the effects between design objects without needing explicit references to those effects.

10.2 Ideation Database System

The Ideation Database system allows the designer to interact with the design elements of the database in two ways. The Database View lets the designer view, edit, add, delete, and search elements in the database. It provides a view into the collective design knowledge of the users of the Ideation Database.

The Analysis View functions as a design worksheet. Once the designer enters a set of functional requirements, the Ideation Database analyzes the information given and searches the database for appropriate design parameters. It arranges those design parameters into sets and presents the sets to the designer along with a design matrix of effects for each set.

10.3 Future Work

The implementation of the Ideation Database system is another step towards the long-range goal of the Axiomatic Design Group - the development of the Thinking
Design Machine. Future work should focus on refining the implementation to provide more analysis and less intrusion during the design process.

10.3.1 Interaction between the Analysis View and the Database View

With the current implementation of the Ideation Database, if a designer does not find an appropriate functional requirement in the Analysis View, he would need to switch to the Database View to add one. Rather than interrupt the flow of the design process in such a manner, the Ideation Database should move the user between the views automatically when necessary. If one of the designer’s functional requirements does not have an associated design parameter, the software should prompt the designer to enter one. In future implementations, the functions of the Database View should by and large be subsumed into the Analysis View. The Database View should exist only as a reference view. There should be a seamless interface to all necessary functions.

10.3.2 Matrix Analysis

The function of the design matrices in the current implementation of the Ideation Database is to provide the designer with information regarding the coupling between functional requirements. No manipulation or analysis is done to the matrices. Those functions are provided in the original axiomatic design software described in [6]. Future versions of the Ideation Database should incorporate those functions. The software should attempt to rearrange the matrices and then present them to the designer as
uncoupled, decoupled, or coupled. The software should also sort the matrices so the uncoupled ones are shown first, followed by the decoupled and then the coupled matrices.

10.3.3 Integration with Other Software Tools

Another area for future work is integrating the Ideation Database with other software tools. This includes the current axiomatic design software as well as other CAD tools such as Pro-Engineer and AutoCAD. With so many proprietary standards, this can be quite a daunting task. However, with the emergence of middleware standards like OLE and CORBA, the task of translating data between the different applications is becoming automated. All the applications developer will have to do is code the interfaces.
References


Appendix - Code

This appendix includes some selected portions of code from the files for the Ideation Database.

A1. Class List

This a list of the Visual C++ classes created and used in the development of the Ideation Database.

CAboutDlg - controls the About dialog box
CAnalysisView - the primary view class, maintains the Analysis View functions and UI callbacks
CAAttributeAlert - controls the Attribute Alert box
CAAttributeDialog - controls the Attribute dialog box used for attribute add and edit
CAAttributeSet - defines the interface to an Attribute object in the database
CDPDialog - controls the DP edit, view dialog box of the Database View
CDPSet - defines the interface to a DP object in the database
CFRSelectDialog - controls the FR selection dialog box of the Analysis View
CFRSet - defines the interface to an FR object in the database
CIDatabaseApp - the main application class, handles initialization of the application
CIDatabaseDoc - the document class, maintains the links to the database

CIDatabaseView - the secondary view class, handles maintenance of design objects

CMainFrame - handles initialization of views and specifies the appearance of the main window

CSearchAlert - controls the Search Alert dialog

CSearchDialog - controls the Search Dialog box in the Database View

A2. CAnalysisView - AnalysisView.cpp

```cpp
void CAnalysisView::OnSearch()
{
    CIDatabaseDoc* pDoc = (CIDatabaseDoc*)GetDocument();
    CFRSelectDialog dlg;
    CString str;
    long count;

    // Getting the analysis form data
    UpdateData();
    UpdateData();

    // Refreshing the recordset to incorporate changes
    m_FRSet.m_strSort = "";
    m_FRSet.m_strFilter = "";
    m_FRSet.Requery();

    // Parse through all records, updating the rank
    while (!m_FRSet.IsEOF())
    {
        // Note : negative values are used to match SQL sort methods
        // which use an ascending ordering.
        count = 0L;
        count -= Matches(m_FREntry, m_FRSet.m_Name);

        // Updating the rank value in the database
        m_FRSet.Edit();
        m_FRSet.m_Rank = count;
        m_FRSet.Update();
        m_FRSet.MoveNext();
    }

    // Sort and requery the FR recordset
    m_FRSet.m_strSort = "Rank";
    m_FRSet.m_strFilter.Format("Rank < %ld", 0L);
```
m_FRSet.Requery();

// Initialize the dialog data members
dlg.m_pSet = &m_FRSet;
dlg.m_searchStr = m_FRentry;

// Invoke the dialog box
if (dlg.DoModal() == IDOK)
{
    str = dlg.m_selectedFR;
    m_ctlFRList.InsertString(-1, str);
}

// Refreshing the recordset
m_FRSet.m_strFilter = "";
m_FRSet.Requery();

void CAnalysisView::OnAnalyze()
{
    CString str, str2;
    int i, n;
    int LBsize = m_ctlFRList.GetCount();

    // Filling the m_arFR_ID array
    for (i = 0; i < LBsize; i++)
    {
        m_ctlFRList.GetText(i, str2);
        if (str2.Find(']') != -1)
        {
            n = str2.GetLength() - str2.Find(']') - 2;
            str2 = str2.Right(n);
        }
        str.Format("Name = '%s'", str2);
        m_FRSet.FindFirst(str);
        m_arFR_ID[i] = m_FRSet.m_FR_ID;
    }
    m_arFR_ID[LBsize] = -1;

    // Filling the m_arDPCombo array
    GenerateCombinations(LBsize);

    // Setting m_arDP_ID to first DM
    m_currentMatrix = 0;
    m_arDP_ID = m_arDPCombo[m_currentMatrix];

    LoadFRList();
    LoadDPList();
    LoadMatrix();
}
void CAnalysisView::OnPrevMatrix()
{
    // Setting m_arDP_ID to previous DM
    if (m_currentMatrix == 0)
        return;
    else
    {
        m_currentMatrix--;
        m_arDP_ID = m_arDPCombo[m_currentMatrix];
    }

    // Reloading the listboxes
    LoadFRList();
    LoadDPList();
    LoadMatrix();
}

void CAnalysisView::OnNextMatrix()
{
    int i, count, FRcnt;
    FRcnt = m_ctlFRList.GetCount();

    // Determining if there is a next matrix
    count = 0;
    for (i = 0; i < FRcnt; i++)
    {
        if (m_arDPCombo[m_currentMatrix+1][i] == -1)
            count++;
    }
    if (count == FRcnt)
        return;

    // Setting m_arDP_ID to next DM
    m_currentMatrix++;
    m_arDP_ID = m_arDPCombo[m_currentMatrix];

    // Reloading the listboxes
    LoadFRList();
    LoadDPList();
    LoadMatrix();
}

long CAnalysisView::Matches(CString tokens, CString sent)
{
    long count = 0L;
    CString curtok, curword, restok, resword;

    // Handling pathological case
    if (tokens.IsEmpty() || sent.IsEmpty())
        return count;

    // Adding a whitespace character for Find
restok = tokens + " ";

while (!restok.IsEmpty())
{
    // Getting the current token
    curtok = restok.SpanExcluding(" ");
    curtok.MakeUpper();

    // Resetting the word list
    resword = sent + " ";

    while (!resword.IsEmpty())
    {
        // Getting the first word
        curword = resword.SpanExcluding(" ");
        curword.MakeUpper();

        // Checking for a match
        if (curword.Find(curtok) != -1)
            count += 1L;

        // Getting the next word
        if (resword.Find(' ') == -1)
            resword.Empty();
        else
            resword = resword.Right(resword.GetLength() - resword.Find(' ') - 1);
    }

    // Getting the next token
    if (restok.Find(' ') == -1)
        restok.Empty();
    else
        restok = restok.Right(restok.GetLength() - restok.Find(' ') - 1);
}

return count;

void CAnalysisView::OnDeleteFR()
{
    m_ctlFRList.DeleteString(m_ctlFRList.GetCurSel());
}

void CAnalysisView::OnInitialUpdate()
{
    CIDatabaseDoc* pDoc = (CIDatabaseDoc*) GetDocument();
    CFormView::OnInitialUpdate();

    // Setting up the various recordsets
    m_FRSet.m_pDatabase = pDoc->GetDatabase();
}
m_DPSet.m_pDatabase = pDoc->GetDatabase();

try
{
    m_FRSet.Open();
    m_DPSet.Open();
}
catch (CDaoException* e)
{
    AfxMessageBox(e->m_pErrorInfo->m_strDescription);
    e->Delete();
    return;
}

void CAnalysisView::LoadDPList()
{
    CString str, str2;
    int i = 0;

    // Query the m_DPSets to incorporate new DPs in the database
    m_DPSet.Requery();

    // Clear the listbox first
    m_ctlDPList.ResetContent();

    while (m_arFR_ID[i] != -1)
    {
        str.Format("DPID = %ld", m_arDPID[i]);
        if (m_DPSet.FindFirst(str) == 0)
            str2.Format("[DP%d] No DP", i+1);
        else
            str2.Format("[DP%d] %s", i+1, m_DPSet.m_Name);
        m_ctlDPList.InsertString(i, str2);
        i++;
    }
}

void CAnalysisView::LoadMatrix()
{
    CString str, ISstr, FSstr, STstr, row;
    int i, j;
    CString test;
    int size = m_ctlFRList.GetCount();

    m_ctlMatrixList.ResetContent();

    for (i = 0; i < size; i++)
    {
        str.Format("FR_ID = %ld", m_arFR_ID[i]);
        m_FRSet.FindFirst(str);
ISstr = m_FRSet.m_InitialState;
FSstr = m_FRSet.m_FinalState;

row = "";
for (j = 0; j < size; j++)
{
    str.Format("DP_ID = %ld", m_arDP_ID[j]);
    if (m_DPSet.FindFirst(str) == 0)
        row += " ?";
    else
    {
        STstr = m_DPSet.m_StateTransform;

        // Making the comparison
        if (Matches(STstr, ISstr) || Matches(STstr, FSstr))
            row += " X";
        else
            row += " O";
    }
}
m_ctlMatrixList.InsertString(i, row);

if (size != 0)
{
    str.Format("Design Matrix %d", m_currentMatrix);
    m_ctlMatrixList.InsertString(i, "");
    m_ctlMatrixList.InsertString(i+1, str);
}

void CAnalysisView::LoadFRList()
{
    CString str, str2;
    int i = 0;

    // Clear the listbox first
    m_ctlFRList.ResetContent();

    while (m_arFR_ID[i] != -1)
    {
        str.Format("FR_ID = %ld", m_arFR_ID[i]);
        m_FRSet.FindFirst(str);
        str2.Format("[FR%d] %S", i+1, m_FRSet.m_Name);
        m_ctlFRList.InsertString(i, str2);
        i++;
    }
}

void CAnalysisView::GenerateCombinations(int FRcnt)
{
    // Reinitializing the m_arDPCombo array

    // Code continuation...
ClearComboArray();

// Filling in the rows recursively
RecursiveCombo(FRcnt, FRcnt-1, 1);

void CAnalysisView::ClearComboArray()
{
    int i, j;

    for (i = 0; i < 10; i++)
    {
        for (j = 0; j < 100; j++)
        {
            m_arDPCombo[i][j] = -1;
        }
    }
}

void CAnalysisView::RecursiveCombo(int FRcnt, int col, int n)
{
    // Function to fill in row
    CString str;
    int i, j, k, result;
    BOOL flag_no_dp = FALSE;

    // Base case
    if (col == -1)
        return;

    // Filling in the next row
    k = 0;
    str.Format("FR_ID = %ld", m_arFR_ID[col]);
    result = m_DPSet.FindFirst(str);

    if (result == 0)
    {
        m_arDPCombo[k][col] = -1;
        flag_no_dp = TRUE;
    }

    while (result != 0)
    {
        for (i = 0; i < n; i++)
        {
            m_arDPCombo[k][col] = m_DPSet.m_DP_ID;
            k++;
        }
        result = m_DPSet.FindNext(str);
    }
}
// Continue filling in the previous rows
for (i = 1; i < FRcnt-col; i++)
{
    for (j = 0; j < k-n; j++)
    {
        m_arDPCombo[n+j][col+i] = m_arDPCombo[j][col+i];
    }
}

// Recursive call
if (flag_no_dp)
    RecursiveCombo(FRcnt, col-1, n); // No DP for this FR
else
    RecursiveCombo(FRcnt, col-1, k);

void CAnalysisView::OnClearFrs()
{
    m_ctlFRList.ResetContent();
    OnAnalyze();
}

A3. CDPDialog - DPDialog.cpp

void CDPDialog::OnDeleteAttribute()
{
    CString str, name;

    // Check if we are in the midst of adding a new record
    if (m_bAddMode)
        return;

    // Getting the attribute name
    m_ctlAttributeList.GetText(m_ctlAttributeList.GetCurSel(), str);
    name = str.Left(str.Find(' '));

    // Check if a recordset is open already. If so, close it.
    if (m_attributeSet.IsOpen())
    {
        m_attributeSet.Close();
    }

    m_attributeSet.m_strFilter.Format("AttributeName = '%s'", (LPCSTR) name);
try {
    m_attributeSet.Open();
}
catch (CDaoException* e) {
    AfxMessageBox(e->m_pErrorInfo->m_strDescription);
e->Delete();
    return;
}

try {
    m_attributeSet.Delete();
}
catch (CDaoException* e) {
    AfxMessageBox(e->m_pErrorInfo->m_strDescription);
e->Delete();
    return;
}

// Refresh the attributes list
LoadListBox();

void CDPDialog::OnAddAttribute()
{
    CAttributeDialog dlg;
    CAttributeAlert alert;

    // Check if we are in the midst of adding a new record
    if (mbAddMode)
        return;

    // Check if a recordset is open already. If not, open one first.
    if (!m_attributeSet.IsOpen())
    {
        try {
            m_attributeSet.Open();
        }
        catch (CDaoException* e) {
            AfxMessageBox(e->m_pErrorInfo->m_strDescription);
e->Delete();
            return;
        }
    }

    m_attributeSet.AddNew();
}
// Invoke the dialog box
if (dlg.DoModal() == IDOK)
{
    // retrieve the dialog data
    m_attributeSet.m_OwnerID = m_DP_ID;
    m_attributeSet.m_OwnerType = _T("DP");

    // Checking for errors in name field
    if ((dlg.m_Name == "") || (dlg.m_Name.Find(' ') != -1))
    {
        alert.DoModal();
        return;
    }
    else
    {
        m_attributeSet.m_AttributeName = dlg.m_Name;

        m_attributeSet.m_Value = dlg.m_Value;
        m_attributeSet.m_Type = dlg.m_Type;
    }
    m_attributeSet.Update();

    // Refresh the attributes list
    LoadListBox();
}

void CDPDialog::OnEditAttribute()
{
    CString str, name;
    CAttributeDialog dlg;
    CAttributeAlert alert;

    // Check if we are in the midst of adding a new record
    if (m_bAddMode)
        return;

    // Getting the attribute name
    m_ctlAttributeList.GetText(m_ctlAttributeList.GetCurSel(), str);
    name = str.Left(str.Find(' '));

    // Check if a recordset is open already. If so, close it.
    if (m_attributeSet.IsOpen())
    {
        m_attributeSet.Close();
    }

    m_attributeSet.m_strFilter.Format("AttributeName = '%s'", (LPCSTR) name);

    try
    {
        m_attributeSet.Open();
    }
    catch (CDaoException* e)
    {

AfxMessageBox(e->m_pErrorInfo->m_strDescription);
e->Delete();
return;
}

// Initialize dialog data
dlg.m_Name = m_attributeSet.m_AttributeName;
dlg.m_Value = m_attributeSet.m_Value;
dlg.m_Type = m_attributeSet.m_Type;

m_attributeSet.Edit();

// Invoke the dialog box
if (dlg.DoModal() == IDOK)
{
    // retrieve the dialog data
    m_attributeSet.m_OwnerID = m_DPID;
    m_attributeSet.m_OwnerType = _T("DP");

    // Checking for errors in name field
    if ((dlg.m_Name == "") || (dlg.m_Name.Find(' ') != -1))
    {
        alert.DoModal();
        return;
    }
    else
    
    m_attributeSet.m_AttributeName = dlg.m_Name;

    m_attributeSet.m_Value = dlg.m_Value;
    m_attributeSet.m_Type = dlg.m_Type;
}
m_attributeSet.Update();

// Refresh the attributes list
LoadListBox();

void CDPDialog::LoadListBox()
{
    // Check if recordset is open already. If so, close it first.
    if (m_attributeSet.IsOpen())
    {
        m_attributeSet.Close();
    }

    m_attributeSet.m_strFilter.Format("OwnerID = %ld AND OwnerType = "DP",m_DPID);

    try
    {
        m_attributeSet.Open();
    }
catch (CDaoException* e)
{
    AfxMessageBox(e->m_pErrorInfo->m_strDescription);
    e->Delete();
    return;
}

m_ctlAttributeList.ResetContent();
if (m_attributeSet.IsOpen())
{
    while (!m_attributeSet.IsEOF())
    {
        m_ctlAttributeList.AddString(m_attributeSet.m_AttributeName + " " + m_attributeSet.m_Value + " " + m_attributeSet.m_Type);
        m_attributeSet.MoveNext();
    }
}  // attr closed by CDaoRecordset destructor

BOOL CDPDialog::OnInitDialog()
{
    BOOL result = CDialog::OnInitDialog();
    m_attributeSet.m_pDatabase = m_pDatabase;
    LoadListBox();
    return result;
}

void CDPDialog::OnPrevDp()
{
    // Check if we are in the midst of adding a new record
    if (m_bAddMode)
        return;

    // TODO : Update data first
    m_pSet->MovePrev();
    if (m_pSet->IsBOF())
        m_pSet->MoveFirst();
    UpdateDialogData();
}

void CDPDialog::OnNextDp()
{
    // Check if we are in the midst of adding a new record
if (m_bAddMode)
    return;

// TODO : Update data first
m_pSet->MoveNext();

if (m_pSet->isEOF())
    m_pSet->MoveLast();

UpdateDialogData();

void CDPDialog::OnDeleteDp()
{
    // Check if we are in the midst of adding a new record
    if (m_bAddMode)
        return;

    try
    {
        // Delete the Attributes
        m_attributeSet.m_strFilter.Format("OwnerID = %ld AND OwnerType = 'DP'\~m_DP_ID\)
        m_attributeSet.Requery();
        while (!m_attributeSet.IsEOF())
        {
            m_attributeSet.Delete();
            m_attributeSet.MoveNext();
        }

        // Delete the DP
        m_pSet->Delete();
    }
    catch(CDaoException* e)
    {
        AfxMessageBox(e->m_pErrorInfo->m_strDescription);
        e->Delete();
    }

    // Move to the next record after the one just deleted
    m_pSet->MoveNext();
    // If we moved off the end of file, move back to last record
    if (m_pSet->isEOF())
        m_pSet->MoveLast();
    // If the recordset is now empty, clear the fields left over
    // from the deleted record
    if (m_pSet->isBOF())
        m_pSet->SetFieldNull(NULL);

    UpdateDialogData();
}
void CDPDialog::UpdateDialogData()
{
    m_pSet->Edit();

    // Updating the dialog data
    m_Name = m_pSet->m_Name;
    m_StateTransform = m_pSet->m_StateTransform;
    m_Annotation = m_pSet->m_Annotation;
    m_DP_ID = m_pSet->m_DP_ID;

    UpdateData(FALSE);
    LoadListBox();
}

A4. CFRSelectDialog - FRSelectDialog.cpp

void CFRSelectDialog::LoadSelectionBox()
{
    int count = 0;

    // Make sure recordset is open already
    if (!m_pSet->IsOpen())
        return;

    m_ctlSelectionList.ResetContent();
    while (!m_pSet->IsEOF() && (count < 5))
    {
        m_ctlSelectionList.InsertString(count, m_pSet->m_Name);
        m_pSet->MoveNext();
        count++;
    }
}

A5. CIDatabaseDoc - IDatabaseDoc.cpp

BOOL CIDatabaseDoc::OnNewDocument()
{
    if (!m_database.IsOpen())
    {
        // First assume axiom.mdb file is in current directory
        try
        {
            m_database.Open(_T("axiom.mdb"));
        }
        catch (CDaoException* e)
void CIDatabaseView::OnInitialUpdate()
{
    // Linking auxiliary recordsets to primary database
    CIDatabaseDoc* pDoc = GetDocument();
    m_pSet = &pDoc->m_FRSet;
    m_pSet->m_pDatabase = pDoc->GetDatabase();
    if (!m_pSet->m_pDatabase->IsOpen())
        return;

    pDoc->m_attributeSet.m_pDatabase = m_pSet->m_pDatabase;
    pDoc->m_DPSet.m_pDatabase = m_pSet->m_pDatabase;
}

A6. CIDatabaseView - IDatabaseView.cpp
// Calling parent function
CDaoRecordView::OnInitialUpdate();

// Loading the attributes listbox with initial info
LoadListBox();
}

void CIDatabaseView::LoadListBox()
{
    CIDatabaseDoc* pDoc = GetDocument();

    // Check if recordset is open already. If so, close it first.
    if (pDoc->m_attributeSet.IsOpen())
    {
        pDoc->m_attributeSet.Close();
    }

    pDoc->m_attributeSet.m_strFilter.Format("OwnerID = %ld AND OwnerType = 'FR'",
        m_pSet->m_FR_ID);
    try
    {
        pDoc->m_attributeSet.Open();
    }
    catch (CDaoException* e)
    {
        AfxMessageBox(e->m_pErrorInfo->m_strDescription);
        e->Delete();
        return;
    }

    m_ctlAttributeList.ResetContent();
    if (pDoc->m_attributeSet.IsOpen())
    {
        while (!pDoc->m_attributeSet.IsEOF())
        {
            m_ctlAttributeList.AddString(pDoc->m_attributeSet.m_AttributeName
                + " " + pDoc->m_attributeSet.m_Value
                + " " + pDoc->m_attributeSet.m_Type);
            pDoc->m_attributeSet.MoveNext();
        }
    }
    // attr closed by CDaoRecordset destructor
}

BOOL CIDatabaseView::OnMove(UINT nIDMoveCommand)
{
    if (m_bAddMode)
    {
        if (!UpdateData())
        {
            LoadListBox();
        }
    }
return FALSE;
}
try
{
    m_pSet->Update();
}
catch(CDaoException* e)
{
    AfxMessageBox(e->m_pErrorInfo->m_strDescription);
    e->Delete();
}
try
{
    m_pSet->Requery();
    UpdateData(FALSE);
    m_bAddMode = FALSE;
    LoadListBox();
    return TRUE;
}
} 

BOOL result = CDaoRecordView::OnMove(nIDMoveCommand);
LoadListBox();
return result;

void CIDatabaseView::OnRecordAdd()
{
    // If already in Add Mode, complete the previous new record
    if (m_bAddMode)
    
        OnMove(ID_RECORD_FIRST);

    // Clear list box
    m_ctlAttributeList.ResetContent();

    m_pSet->AddNew();
    m_bAddMode = TRUE;
    UpdateData(FALSE);
}

void CIDatabaseView::OnRecordDelete()
{
    CIDatabaseDoc* pDoc = GetDocument();
    try
    {
        if (pDoc->m_DPSet.IsOpen() && pDoc->m_DPSet.CanRestart())
        { 
            // Delete the DPs
            pDoc->m_DPSet.m_strFilter.Format("FR_ID = %ld", m_pSet->m_FRID);
            pDoc->m_DPSet.Requery();

            while (!pDoc->m_DPSet.IsEOF())
            { 
                // Delete the DP Attributes
            }
        }
    }
}

108
pDoc->m_attributeSet.m_strFilter.Format("OwnerID = %ld AND OwnerType = 'DP'", pDoc->m_DPSet.m_ID);
pDoc->m_attributeSet.Requery();
while (!pDoc->m_attributeSet.IsEOF())
{
    pDoc->m_attributeSet.Delete();
    pDoc->m_attributeSet.MoveNext();
}
pDoc->m_DPSet.Delete();
pDoc->m_DPSet.MoveNext();

// Delete the FR Attributes
pDoc->m_attributeSet.m_strFilter.Format("OwnerID = %ld AND OwnerType = 'FR'", m_pSet->m_FR_ID);
pDoc->m_attributeSet.Requery();
while (!pDoc->m_attributeSet.IsEOF())
{
    pDoc->m_attributeSet.Delete();
    pDoc->m_attributeSet.MoveNext();
}

// Delete the FR
m_pSet->Delete();
}
catch(CDaoException* e)
{
    AfxMessageBox(e->m_pErrorInfo->m_strDescription);
    e->Delete();
}

    // Move to the next record after the one just deleted
    m_pSet->MoveNext();
    // If we moved off the end of file, move back to last record
    if (m_pSet->IsEOF())
        m_pSet->MoveLast();
    // If the recordset is now empty, clear the fields left over
    // from the deleted record
    if (m_pSet->IsBOF())
        m_pSet->SetFieldNull(NULL);
    UpdateData(FALSE);
    LoadListBox();
}

void CIDatabaseView::OnRecordRefresh()
{
    if (m_bAddMode)
    {
        m_pSet->CancelUpdate();
        m_pSet->Move(0);
        m_bAddMode = FALSE;
    }
void CIDatabaseView::OnAddAttribute()
{
    CAtributeDialog dlg;
    CAtributeAlert alert;
    CIDatabaseDoc* pDoc = GetDocument();

    // Check if we are in the midst of adding a new record
    if (m_bAddMode)
    {
        OnMove(ID_RECORD_FIRST);
        OnMove(ID_RECORD_LAST);
    }

    // Check if a recordset is open already. If not, open one first.
    if (!pDoc->m_attributeSet.IsOpen())
    {
        try
        {
            pDoc->m_attributeSet.Open();
        }
        catch (CDaoException* e)
        {
            AfxMessageBox(e->m_pErrorInfo->m_strDescription);
            e->Delete();
            return;
        }
    }

    pDoc->m_attributeSet.AddNew();

    // Invoke the dialog box
    if (dlg.DoModal() == IDOK)
    {
        // retrieve the dialog data
        pDoc->m_attributeSet.m_OwnerID = m_pSet->m_FR_ID;
        pDoc->m_attributeSet.m_OwnerType = _T("FR");

        // Checking for errors in name field
        if (dlg.m_Name == "") || (dlg.m_Name.Find(' ') != -1))
        {
            alert.DoModal();
            return;
        } else
        {
            pDoc->m_attributeSet.m_AttributeName = dlg.m_Name;
            pDoc->m_attributeSet.m_Value = dlg.m_Value;
        }
    }
pDoc->m_attributeSet.m_Type = dlg.m_Type;

pDoc->m_attributeSet.Update();

// Refresh the attributes list
LoadListBox();

void CIDatabaseView::OnDeleteAttribute()
{
    CString str, name;
    CIDatabaseDoc* pDoc = GetDocument();

    // Check if we are in the midst of adding a new record
    if (m_bAddMode)
    {
        OnMove(ID_RECORD_FIRST);
        OnMove(ID_RECORD_LAST);
    }

    // Getting the attribute name
    m_ctlAttributeList.GetText(m_ctlAttributeList.GetCurSel(), str);
    name = str.Left(str.Find(' '));

    // Check if a recordset is open already. If so, close it.
    if (pDoc->m_attributeSet.IsOpen())
    {
        pDoc->m_attributeSet.Close();
    }

    pDoc->m_attributeSet.m_strFilter.Format("AttributeName = '%s'", (LPCSTR) name);

    try
    {
        pDoc->m_attributeSet.Open();
    }
    catch (CDaoException* e)
    {
        AfxMessageBox(e->m_pErrorInfo->m_strDescription);
        e->Delete();
        return;
    }

    try
    {
        pDoc->m_attributeSet.Delete();
    }
    catch (CDaoException* e)
    {
        AfxMessageBox(e->m_pErrorInfo->m_strDescription);
        e->Delete();
        return;
    }
}
// Refresh the attributes list
LoadListBox();

void CIDatabaseView::OnEditAttribute()
{
    CString str, name;
    CIDatabaseDoc* pDoc = GetDocument();
    CAttributeDialog dlg;
    CAttributeAlert alert;

    // Check if we are in the midst of adding a new record
    if (m_bAddMode)
    {
        OnMove(ID_RECORD_FIRST);
        OnMove(ID_RECORD_LAST);
    }

    // Getting the attribute name
    m_ctlAttributeList.GetText(m_ctlAttributeList.GetCurSel(), str);
    name = str.Left(str.Find(' '));

    // Check if a recordset is open already. If so, close it.
    if (pDoc->m_attributeSet.IsOpen())
    {
        pDoc->m_attributeSet.Close();
    }

    pDoc->m_attributeSet.m_strFilter.Format("AttributeName = '%s'", (LPCSTR) name);
    try
    {
        pDoc->m_attributeSet.Open();
    }
    catch (CDaoException* e)
    {
        AfxMessageBox(e->m_pErrorInfo->m_strDescription);
        e->Delete();
        return;
    }

    // Initialize dialog data
    dlg.m_Name = pDoc->m_attributeSet.m_AttributeName;
    dlg.m_Value = pDoc->m_attributeSet.m_Value;
    dlg.m_Type = pDoc->m_attributeSet.m_Type;

    pDoc->m_attributeSet.Edit();

    // Invoke the dialog box
    if (dlg.DoModal() == IDOK)
    {
        
}

112
// retrieve the dialog data
pDoc->m_attributeSet.m_OwnerID = m_pSet->m_FR_ID;
pDoc->m_attributeSet.m_OwnerType = _T("FR");

// Checking for errors in name field
if ((dlg.m_Name == "") || (dlg.m_Name.Find(' ') != -1))
{
    alert.DoModal();
    return;
}
else
    pDoc->m_attributeSet.m_AttributeName = dlg.m_Name;

    pDoc->m_attributeSet.m_Value = dlg.m_Value;
pDoc->m_attributeSet.m_Type = dlg.m_Type;
}
pDoc->m_attributeSet.Update();

// Refresh the attributes list
LoadListBox();
}

void CIDatabaseView::OnViewDp()
{
    CIDatabaseDoc* pDoc = GetDocument();
    CDPDialog dlg;

    // Check if a recordset is open already. If so, close it.
    if (pDoc->m_DPSet.IsOpen())
    {
        pDoc->m_DPSet.Close();
    }

    pDoc->m_DPSet.m_strFilter.Format("FR_ID = %ld", m_pSet->m_FR_ID);
    try
    {
        pDoc->m_DPSet.Open();
    }
    catch (CDaoException* e)
    {
        AfxMessageBox(e->m_pErrorInfo->m_strDescription);
        e->Delete();
        return;
    }

    // Initialize dialog data
    dlg.m_AssociatedFR = m_pSet->m_Name;
    dlg.m_Name = pDoc->m_DPSet.m_Name;
    dlg.m_StateTransform = pDoc->m_DPSet.m_StateTransform;
    dlg.m_Annotation = pDoc->m_DPSet.m_Annotation;
    dlg.m_DP_ID = pDoc->m_DPSet.m_DP_ID;
dlg.m_pDatabase = m_pSet->m_pDatabase;
dlg.m_pSet = &pDoc->m_DPSet;

pDoc->m_DPSet.Edit();

// Invoke the dialog box
if (dlg.DoModal() == IDOK)
{
    pDoc->m_DPSet.m_Name = dlgl.m_Name;
    pDoc->m_DPSet.m_StateTransform = dlgl.m_StateTransform;
    pDoc->m_DPSet.m_Annotation = dlgl.m_Annotation;
}
pDoc->m_DPSet.Update();

void CIDatabaseView::OnAddDp()
{
    CDPDialog dlg;
    CIDatabaseDoc* pDoc = GetDocument();

    // Check if we are in the midst of adding a new record
    if (m_bAddMode)
    {
        OnMove(ID_RECORD_FIRST);
        OnMove(ID_RECORD_LAST);
    }

    // Check if a recordset is open already. If not, open one first.
    if (!pDoc->m_DPSet.IsOpen())
    {
        try
        {
            pDoc->m_DPSet.Open();
        }
        catch (CDaoException* e)
        {
            AfxMessageBox(e->m_pErrorInfo->m_strDescription);
            e->Delete();
            return;
        }
    }

dlg.m_AssociatedFR = m_pSet->m_Name;
dlg.m_pDatabase = m_pSet->m_pDatabase;
dlg.m_bAddMode = TRUE;
pDoc->m_DPSet.AddNew();

// Invoke the dialog box
if (dlg.DoModal() == IDOK)
{
    pDoc->m_DPSet.m_FR_ID = m_pSet->m_FR_ID;
    pDoc->m_DPSet.m_Name = dlgl.m_Name;
}
pDoc->m_DPSet.m_StateTransform = dlg.m_StateTransform;
pDoc->m_DPSet.m_Annotation = dlg.m_Annotation;
}
dlg.m_bAddMode = FALSE;
pDoc->m_DPSet.Update();
}

void CIDatabaseView::OnSearchFRs()
{
    CSearchDialog dlg;
    CSearchAlert alert;
    CString FRstr, ISstr, FSstr, ATstr, attr;
    CAttributeSet attrSet;
    long threshold, count;

    // Check if we are in the midst of adding a new record
    if (m_bAddMode)
        OnMove(ID_RECORD_FIRST);
    // Invoke the dialog box
    if (dlg.DoModal() == IDOK)
    {
        FRstr = dlg.m_FRName;
        ISstr = dlg.m_InitialState;
        FSstr = dlg.m_FinalState;
        ATstr = dlg.mAttributes;
        threshold = -1 * dlg.m_threshold;  // negative value due to SQL sort method
    }

    // Requery the existing recordset (which may be filtered)
    m_pSet->m_strSort = "";
    m_pSet->m_strFilter = "";

    try
    {
        m_pSet->Requery();
    }
    catch (CDaoException* e)
    {
        AfxMessageBox(e->m_pErrorInfo->m_strDescription);
        e->Delete();
        return;
    }

    // Parse through all records, updating the rank
    while (!m_pSet->IsEOF())
    {
        // Note: negative values are used to match SQL sort methods
        // which use an ascending ordering.
        count = 0L;
        count -= Matches(FRstr, m_pSet->m_Name);
count -= Matches(ISstr, m_pSet->m_InitialState);
count -= Matches(FSstr, m_pSet->m_FinalState);

// Building the attribute string for search
attr = "";
attrSet.m_strFilter.Format("OwnerID = %ld AND OwnerType = 'FR'", m_pSet->m_FR_ID);
attrSet.m_pDatabase = m_pSet->m_pDatabase;
attrSet.Open();
while (!attrSet.IsEOF())
{
    attr += attrSet.m_AttributeName;
    attr += " ";
    attrSet.MoveNext();
}
attrSet.Close();
attr.TrimRight();
count -= Matches(ATstr, attr);

// Updating the rank value in the database
m_pSet->Edit();
m_pSet->m_Rank = count;
m_pSet->Update();
m_pSet->MoveNext();

// Requery and sort the FR recordset
m_pSet->m_strSort = "Rank";
m_pSet->m_strFilter.Format("Rank < %ld", threshold);
m_pSet->Requery();
if (m_pSet->IsEOF())
{
    alert.DoModal();
    m_ctlAttributeList.ResetContent();
}
UpdateData(FALSE);
LoadListBox();

long CIDatabaseView::Matches(CString tokens, CString sent)
{
    long count = 0L;
    CString curtok, curword, restok, resword;

    // Handling pathological case
    if (tokens.IsEmpty() || sent.IsEmpty())
        return count;

    // Adding a whitespace character for Find
    restok = tokens + " ";

while (!restok.IsEmpty())
{
    // Getting the current token
    curtok = restok.SpanExcluding(" ");
    curtok.MakeUpper();

    // Resetting the word list
    resword = sent + " ";

    while (!resword.IsEmpty())
    {
        // Getting the first word
        curword = resword.SpanExcluding(" ");
        curword.MakeUpper();

        // Checking for a match
        if (curword.Find(curtok) != -1)
            count += 1L;

        // Getting the next word
        if (resword.Find(' ') == -1)
            resword.Empty();
        else
            resword = resword.Right(resword.GetLength() - resword.Find(' ') - 1);
    }

    // Getting the next token
    if (restok.Find(' ') == -1)
        restok.Empty();
    else
        restok = restok.Right(restok.GetLength() - restok.Find(' ') - 1);
}

return count;

void CIDatabaseView::OnRecordShowAll()
{
    m_pSet->m_strSort.Empty();
    m_pSet->m_strFilter.Empty();
    m_pSet->Requery();
    UpdateData(FALSE);
    LoadListBox();
}

**A7. CMainFrame - MainFrame.cpp**

void CMainFrame::OnViewDatabase()
void CMainFrame::SwitchToView(eView nView)
{
    CView* pOldActiveView = GetActiveView();
    CView* pNewActiveView = (CView*)GetDlgItem(nView);
    if (pNewActiveView == NULL)
    {
        switch(nView) {
            case ANALYSIS:
                pNewActiveView = (CView*)new CAnalysisView;
                break;
            case DATABASE:
                pNewActiveView = (CView*)new CIDatabaseView;
                break;
        }
    }
    CCreateContext context;
    context.m_pCurrentDoc = pOldActiveView->GetDocument();
    pNewActiveView->Create(NULL, NULL, 0L, CFrameWnd::rectDefault,
                          this, nView, &context);
    pNewActiveView->OnInitialUpdate();
    SetActiveView(pNewActiveView);
    pNewActiveView->ShowWindow(SW_SHOW);
    pOldActiveView->ShowWindow(SW_HIDE);
    pOldActiveView->SetDlgCtrlID(
        pOldActiveView->GetRuntimeClass() ==
        RUNTIME_CLASS(CAnalysisView) ? ANALYSIS : DATABASE);
    pNewActiveView->SetDlgCtrlID(AFX_IDW_PANE_FIRST);
    RecalcLayout();
}

void CMainFrame::OnUpdateViewDatabase(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(!GetActiveView()->IsKindOf(RUNTIME_CLASS(CIDatabaseView)));}

void CMainFrame::OnUpdateViewAnalysis(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(!GetActiveView()->IsKindOf(RUNTIME_CLASS(CAnalysisView)));
}