INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313.761.4700 800.521.0600
An artificial mechanical designer based on an object-oriented approach

Jeong, Taesik, Ph.D.

Case Western Reserve University, 1992

Copyright ©1992 by Jeong, Taesik. All rights reserved.
AN ARTIFICIAL MECHANICAL DESIGNER
BASED ON AN OBJECT-ORIENTED APPROACH

by

TAESIK JEONG

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Thesis Advisor: THOMAS P. KICHER

Department of Mechanical and Aerospace Engineering

CASE WESTERN RESERVE UNIVERSITY

January 1992
Copyright © 1992 by TAESIK JEONG
CASE WESTERN RESERVE UNIVERSITY

GRADUATE STUDIES

We hereby approve the thesis of

Taesik Jeong

candidate for the Ph.D

degree.*

Signed:

[Signatures]

(Chairman)

Date

*We also certify that written approval has been obtained for any proprietary material contained therein.
I grant to Case Western Reserve University the right to use this work, irrespective of any copyright, for the University's own purposes without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.

[Signature]
AN ARTIFICIAL MECHANICAL DESIGNER
BASED ON AN OBJECT-ORIENTED APPROACH

Abstract

by

TAESIK JEONG

An object-oriented programming technique is investigated along with Artificial Intelligence (AI) paradigms, such as artificial neural networks and expert systems, to develop the framework for mechanical design automation systems. A task oriented decomposition approach is also applied to conceptualize the task-performing object (or task-object) in which common behavior and communication protocols are encapsulated. Each task in the entire design process, either controlling design strategies or performing design methods, is made into a task-performing object.

An Artificial Mechanical Designer (AMD) is simply a collection of those task-performing objects, which performs a mechanical component design. Once an AMD is constructed, it is treated as a single task-performing object and shares the same behavior and communication ability in terms of receiving a task-order and requesting a task. The task-performing object and AMD are designed to be compatible with each other, connectable to form more complex task-performing objects, and expandable to enhance their design functionalities.
To emulate design methodologies used by human engineers in the real world practice, various AI applications are employed and made into task-performing objects. They can be furnished with various types of knowledge, including existing design data, formalized knowledge, and heuristic knowledge. Accordingly, the engineer is encouraged to build design systems, applicable to their design assignments, utilizing their own design knowledge. It is possible, since an AMD is a non-prefixed framework which can accept any kind of task-performing objects to form various design systems.

Furthermore, a computer application, the AMD Development Kit (AMDDK), has been implemented to provide a set of tools to the engineer. The AMDDK has pre-built task-performing objects such as design-control objects, method-performing objects, and task-assistant objects, as well as user interface objects. The guidelines specified have been developed for all abstract task-performing objects. To demonstrate the capabilities of the AMD with the task-performing objects, a gear design task is investigated and successfully implemented. The application of a gear AMD provided the following three advantages: 1. Mechanical design can be fully automated utilizing an object-oriented programming along with Artificial Intelligence techniques, 2. Any design systems developed based on the concept of the task-performing object are compatible to each other and interchangeable, 3. Expandability of the existing design systems is only limited by an engineer's design assignments.
Acknowledgements

First of all I would like to express my appreciation to Dr. Thomas P. Kicher whose professional and personal support, encouragement, and invaluable guidance made this work possible. I would also especially like to thank Dr. Ronald J. Zab for his enduring patience and for sharing his knowledge of gear design.

I would like to extend my thanks to Dr. Dwight T. Davy and Dr. Joseph M. Mansour for their invaluable support and specially to Dr. Yoh-Han Pao whose brilliant concept and knowledge of Artificial Intelligence techniques established the foundation of this work. I thank my friend, Jeff Knaus, for sharing his genius knowledge so that I could strengthen my background in object-oriented programming.

Finally, I acknowledge the patience and sacrificial support of three ladies whom I love, my mother, Kookhwa Jeong (Jang), my aunt, Bongsook Hwang (Jeong), and my wife, Jisun Jeong (Kang).
# Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>iv</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td><strong>Chapter 1  Introductions</strong></td>
<td></td>
</tr>
<tr>
<td>1.1 Role of Computers in Engineering Design</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Trend of Current Design Automation Systems</td>
<td>3</td>
</tr>
<tr>
<td>1.3 An Alternate Approach with Artificial Mechanical Designer</td>
<td>5</td>
</tr>
<tr>
<td><strong>Chapter 2  Mechanical Design and Artificial Intelligence</strong></td>
<td></td>
</tr>
<tr>
<td>2.1 Characteristics of Mechanical Design</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Initial Design with Artificial Neural Networks</td>
<td>13</td>
</tr>
<tr>
<td>2.3 Redesign with Knowledge-Based System</td>
<td>18</td>
</tr>
<tr>
<td><strong>Chapter 3  Object-Oriented Approach</strong></td>
<td></td>
</tr>
<tr>
<td>3.1 Object-Oriented Programming</td>
<td>22</td>
</tr>
<tr>
<td>3.2 Object World of Mechanical Design</td>
<td>29</td>
</tr>
<tr>
<td>3.3 Definition of Task-Object</td>
<td>34</td>
</tr>
<tr>
<td>3.4 Developing Subclasses of <em>TaskObject</em> Class</td>
<td>40</td>
</tr>
<tr>
<td><strong>Chapter 4  Implementations of Artificial Mechanical Designer</strong></td>
<td></td>
</tr>
<tr>
<td>4.1 Definition of Artificial Mechanical Designer (AMD)</td>
<td>47</td>
</tr>
<tr>
<td>4.2 Subclasses of <em>TaskAssistant</em> Class</td>
<td>52</td>
</tr>
</tbody>
</table>
4.3 Subclasses of MethodPerformer Class 60
4.4 Subclasses of ControlPerformer Class 69
4.5 Developing Multi-Level AMD Construction 78
4.6 Implementation of AMD Development Kit in Smalltalk 81

Chapter 5 AMD Application to Gear Design

5.1 Organizing Design Procedures 89
5.2 Preparing Design Data for Artificial Neural Network Task-Object 92
5.3 Representing Design Knowledge for Inference Engine Task-Object 99
5.4 Configuring Gear AMD with Selected Task-Objects 102

Chapter 6 Conclusions

6.1 Summary 106
6.2 Limitations of Current Implementation and Future Plans 108

References 112

Appendix A Class Definitions of Artificial Mechanical Designer 118

A.1 Class Hierarchy of Task-Objects 119
A.2 Definitions of AMD Base Classes 120

Appendix B Sample Outputs of Gear AMD 137

B.1 Performance History Outputs 138
B.2 Design Parameter Outputs 143
# Lists of Figures

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Basic Idea of Artificial Mechanical Designer</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>Model of Mechanical Design</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Types of Data Patterns</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>LVQ and GDR</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>Components of Inference Engine</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Components of Object</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Example of Class Hierarchy and Instances of its classes</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>Overwriting or Expanding Superclass Methods</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>Expanding Class with Subclasses</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>Mechanism of Creating Instance</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>Object World of Gear Manufacturing Company</td>
<td>30</td>
</tr>
<tr>
<td>3.7</td>
<td>Object World of Gear Design Team</td>
<td>31</td>
</tr>
<tr>
<td>3.8</td>
<td>Object-Oriented Model of Mechanical Design</td>
<td>33</td>
</tr>
<tr>
<td>3.9</td>
<td>Object vs. Task-Object</td>
<td>34</td>
</tr>
<tr>
<td>3.10</td>
<td>Elements of <em>TaskObject</em> class and its Basic Mechanism</td>
<td>36</td>
</tr>
<tr>
<td>3.11</td>
<td>Task Performing Mechanism</td>
<td>38</td>
</tr>
<tr>
<td>3.12</td>
<td>Task Requesting Mechanism</td>
<td>39</td>
</tr>
<tr>
<td>3.13</td>
<td>Base Class Hierarchy of Task-Objects</td>
<td>41</td>
</tr>
<tr>
<td>3.14</td>
<td><em>TaskPerformer</em> Class Hierarchy Example for Gear Design</td>
<td>44</td>
</tr>
<tr>
<td>3.15</td>
<td><em>TaskAssistant</em> Class Hierarchy Example for Gear Design</td>
<td>45</td>
</tr>
</tbody>
</table>
4.1 Structural Model of Artificial Mechanical Designer
4.2 Procedures in task-performer Method of DesignController Class
4.3 Implementation of Geometry Class
4.4 Specific Method Examples in Subclasses of Geometry Class
4.5 Implementation of Rating Class
4.6 Implementation of MaterialProperty Class
4.7 Example of Gear-Geometry Class Hierarchy
4.8 Current Implementation of MethodPerformer Class Hierarchy
4.9 Neural Network Training and Consulting Diagram
4.10 Element Objects in LVQNet and GDRNet Classes
4.11 Implementation of SingleNeuralNet Class
4.12 Implementation of ForwardInference Class
4.13 Control Object vs. Method Object
4.14 Implementation of InitialDesigner Class
4.15 Implementation of ReDesigner Class
4.16 Examples of Rules
4.17 Task Request Between AMDs and Task-Objects
4.18 Design Evaluating Task-Object in Multi-Level AMD
4.19 Example of Multi-Level AMD Hierarchy
4.20 Basic Screen of AMD Development Kit
4.21 Class Registering Dialog Boxes
4.22 User Input Parameter Entry Dialog Box
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.23</td>
<td>Initial Design Group Dialog Box</td>
<td>84</td>
</tr>
<tr>
<td>4.24</td>
<td>User Input Dialog Box</td>
<td>86</td>
</tr>
<tr>
<td>4.25</td>
<td>Design Outputs Displayed in Monitoring Windows</td>
<td>86</td>
</tr>
<tr>
<td>4.26</td>
<td>Relationship between AMDDK and AMD in Performance</td>
<td>87</td>
</tr>
<tr>
<td>5.1</td>
<td>Model of Gear AMD and Design Knowledge Sources</td>
<td>90</td>
</tr>
<tr>
<td>5.2</td>
<td>Original Catalog Training Patterns</td>
<td>93</td>
</tr>
<tr>
<td>5.3</td>
<td>Transformed Training Patterns</td>
<td>93</td>
</tr>
<tr>
<td>5.4</td>
<td>Product Catalog and Data Pattern File Samples</td>
<td>95</td>
</tr>
<tr>
<td>5.5</td>
<td>Format of GDR Net Files</td>
<td>97</td>
</tr>
<tr>
<td>5.6</td>
<td>Sample Smalltalk Code for Selected Methods of GearInitialDesigner</td>
<td>98</td>
</tr>
<tr>
<td>5.7</td>
<td>Sample Rule Tree of Gear Redesign Knowledge</td>
<td>100</td>
</tr>
<tr>
<td>5.8</td>
<td>Sample Smalltalk Code for Rule Methods</td>
<td>101</td>
</tr>
<tr>
<td>5.9</td>
<td>Fully Configured Gear AMD</td>
<td>104</td>
</tr>
</tbody>
</table>
Chapter 1

Introductions

1.1 Role of Computers in Engineering Design

The term, Computer Aided Design (known as CAD) has been widely used among the design industry during the last decade. The role of computers in engineering design has been evolutionarily improved to increase the design productivity as rapidly as the techniques of microcomputers have improved. In the early years, computers were only capable of helping engineers carry out routine, but fairly complicated, calculation tasks. When fine graphics hardware and software became available, computers were used to aid in the generation of drawings, i.e. blue prints. For this reason, the term CAD is often referred to as Computer Aided Drafting. The early to middle 1960s was a fertile period for the early stage CAD systems, such as DAC/1 (Design Augmented by Computer) developed at General Motors [1]. Since then, many commercial versions of CAD software were released and have been successfully utilized in various engineering fields. Some of them support a customizing capability through either built-in macro languages or regular programming languages, such as the C language. This capability enables engineers to tailor and expand functionalities of CAD software in order to meet their task requirements. Furthermore, algorithmic design procedures can be programmed directly into CAD software. During recent years, CAD software gained more powerful features, such as 3-D capability, feature based
design, solid modeling, parametric modeling, and the list goes on. Engineers can obtain both engineering and visualizing power by using computers, thereby realizing the true meaning of Computer Aided Design. In fact, engineering and visualizing are two important aspects of the engineer's design activity that have been influenced by the introduction of computers.

Today, researchers in both academic and industrial environments attempt to develop design automation systems which have human-like intelligence. Their objectives include emulation of the human engineer's cognitive design processes by a computer. These activities are gradually changing the role of computers in engineering fields. Researchers have adopted various Artificial Intelligence (AI) techniques to develop more effective and intelligent design automation systems. Knowledge-based systems, one sub-field of AI, has received wide acceptance in many design areas. Engineers extract their heuristic knowledge or rules of thumb into rule-like computer knowledge represented symbolically. Some languages or production systems, such as the KEE (Knowledge Engineering Environment™) System and OPS5 (Official Production System, Version 5), have been developed exclusively for knowledge-based programming [31]. Such systems may encourage engineers to build design systems furnished with their own knowledge and experience.

Modern computer techniques have also changed the engineer's attitude toward approaching design tasks. Engineers no longer pay attention to the accuracy
of their calculations nor the appearance of their drawings. Instead, they concentrate their attention on improving design methodologies and optimizing problem solutions. Computers are being utilized in almost every aspect of the design process, in one way or another. Now, it is hard to imagine how engineers complete their design assignments without computers.

1.2 Trend of Current Design Automation Systems

Many researchers and engineers in mechanical engineering fields have been involved in developing design automation systems, from simple engineering systems to complex AI based systems. The efforts of various investigations have yielded several effective algorithms which have been proven to perform well in producing the expected output. Most of the current approaches are based on the expert system paradigm, where both symbolic and numeric data processing are employed. Hierarchical (or task oriented) decomposition and automatic design step (or path) generation are perhaps the most common approaches found in mechanical design expert systems. However, a taxonomy of those approaches may not be feasible, since almost every system is developed based on a different hybrid.

In reference [43], the integration of decomposition and path searching is utilized to design hydraulic systems. In references [36] and [37], all components in a parallel axis gear drive system are modularized and a control coordinator is added to supervise each of the sub-modules, which are organized in a hierarchical form. The transmission housing design expert system found in reference [40] serves as a
single module for the above system. Similar work can be found in reference [42], where the Top Level Design Manager is designed to perform as a planning program. In reference [38], the Core Module serves as a central processing unit for the coordination between entire modules (not sub-modules). The Communication Coordinator is employed for a motor/pump design expert system in reference [39]. It is responsible for handling all interactions between the sub-modules. References [47] and [48] demonstrate the use of object-oriented programming techniques in combination with an expert system.

An emerging technology in the AI field is the artificial neural network. In reference [28], existing numeric design data are used to train a multi-level artificial neural networks for an initial gear design task. This area is further explored through the integration of fuzzy sets and neural net computing as introduced in reference [27]. The method is called the Episodal Associative Memory which has demonstrated its ability for retrieval of previous designs on the basis of qualitative geometry. Another application of the artificial neural network is a process fault diagnosis system as shown in reference [26]. The current artificial neural network applications are primarily applied in development of systems that represent, store, and retrieve existing design data.

Although these methodologies demonstrate applicability to mechanical designs, each of them may be truly applicable to only a portion of the entire design process. It is not desirable to bind design processes into a prefixed method which may vary with each individual engineer. Many of these approaches only address
output compatibility and do not resemble the human method of problem solving, namely procedural compatibility. These systems may be more efficient than human engineers in practice. However, if the engineer lacks the familiarity necessary to manipulate the systems, it becomes infeasible for them, as users, to represent their knowledge, experience, or data into the systems as they would wish. For this reason, how the design process is executed and what knowledge is used to accomplish the task became the main consideration of this project.

Another deficiency in current systems is that they are not compatible and can not be connected to build higher level or more complex systems. Almost every system was developed in a different framework. As an example, there are two mechanical component design systems, a gear set design system and a shaft design system. Let's assume both systems are well built and perform each component design as they are intended. However, it may not be possible to utilize them to build a speed reducer design system due to incompatibility in their respective frameworks. Invaluable manpower and time must be wasted to reprogram these component design systems. Nevertheless, assume again that a speed reducer design system can be completed which contains a gear set and a shaft design modules. Later, new shaft design methodology is developed and a new system is built. It should be possible to replace the old shaft design module with a new one without reprogramming. Another goal of this project is to build fully interchangeable and easily expandable design systems.
1.3 An Alternate Approach with Artificial Mechanical Designer

An Artificial Mechanical Designer (AMD) is an object-oriented programming based framework. It uses a task-oriented decomposition approach found in many similar works (see Section 1.2). The main ingredient of AMD are objects, specifically task-performing objects. Each object knows its own task and what data are required to perform it. In addition to that, each object knows how to communicate with other objects by means of receiving task orders and requesting tasks. Any object can be replaced by other objects that performs the same task but possibly use a different design strategy. AMD is primarily designed to carry out a mechanical component design using methods that simulate those used by human engineers. In addition, it can be connected with other AMDs, since AMD itself is also a task-performing object which shares exactly the same framework and communication protocol as other task-performing objects. This is the most fruitful advantage of applying object-oriented programming. Multiple component design AMDs can work together along with other task-performing objects to establish a unit design AMD. This unit design AMD will serve as one of the task-performing objects in an assembled machine design AMD as illustrated in Figure 1.1. The top level AMD is again a task-performing object just like ones in component design AMDs.

AMD can be compared with children's toy blocks. Blocks fit each other no matter how big or small they are, working together to build any imaginable object. A yellow block can be replaced by a red block without rebuilding it. If more
blocks are desired, they can be acquired from any toy store. From the view point of children, they do not make the blocks, but they merely put them together to build a doll house without knowing any expert construction skills. The only input from the children is the effort of putting them together and a little imagination. If a defective block is found, it can be exchanged at the store from which it was
purchased. The conceptual foundation of AMD is based essentially on children's toy blocks. Therefore, let the engineer execute their work using pre-developed task-performing objects and let them furnish a design system with their own knowledge, not the system developer's methods and knowledge.
2.1 Characteristics of Mechanical Design

As with all other engineering design activities, the mechanical design can be described as an iterative procedure based on trial and error. An engineer searches for an acceptable solution to meet design requirements among many possible ones throughout the entire design activity. On the other hand, an engineer also attempts to obtain a refined solution by considering economic factors, inspecting manufacturing feasibility, and contemplating the consistency with past designs. This iterative activity is illustrated in Figure 2.1, which represents one possible model of a mechanical design procedure and is widely referred to by many similar works [38][41][46].

Referring to the figure, an engineer receives a set of user supplied values, including any design constraints, as design requirements and determines values for initial design parameters. The initial design parameters are normally coupled (functionally dependent) to each other, which may only be determined by knowing other coupled parameters through formulas. In extreme cases, parameters must be determined without any known formulas. Therefore, the initial design parameters can be referred to as independent parameters and may influence the entire design process. Design parameters of this type must be selected by an engineer using available resources such as past design experience, accumulated company
Figure 2.1 Model of Mechanical Design
know-how, and an engineer's unique expertise. Formalizing such heuristic knowledge to use in a general design paradigm is infeasible, since it relies heavily on each design strategy and varies by each engineer's design experience. Specifically, an engineer goes through the initial design step only once, which implies the direction of the remaining design. Therefore, the initial design step can be viewed as the classification or recognition of a design pattern restricted by the given conditions.

Once the initial design parameters have been determined, all dependent design parameters can be obtained and evaluated. The design evaluation step can be further detailed into three steps: material property, geometry calculation, and design rating as illustrated in Figure 2.1. The first two steps are necessary to carry out the design rating step. Fortunately, all three steps are usually formalized procedures, which are readily available to an engineer to utilize. Thus, no special knowledge or experience is required to complete these steps although there exist a few exceptions. If the rating of a proposed design is within the desired range (acceptable range), the evaluation is said to be successful and the design goal has been achieved. These evaluation procedures can be easily computerized using conventional programming techniques. In fact, many computer versions of design evaluating programs for selected components are available commercially. Once the design goal is achieved, the optional design can be performed. In this step an engineer determines all evaluation-independent parameters, then the design output is finally obtained.
When the evaluation indicates a failed design, an engineer analyzes the reasons of failure and examines what parameters can be modified. In this step, an engineer makes every effort to accomplish the design successfully, by carefully selecting one or more design parameters and determining the amount to be changed. This step is achieved by an engineer's redesign knowledge, containing rules of thumb, common sense, and casual knowledge, which have been unconsciously accumulated and remembered throughout the past design experiences. Such knowledge is deeply related to the engineer's cognitive and intuitive abilities and are referred to as decision making knowledge. An engineer with greater design knowledge and experience will produce a better decision and as a result, shorten the number of redesign iterations. However, such decision making knowledge can not be generalized and a unique method can not be derived. It is even dangerous to attempt to bind such knowledge within certain forms. The process goes back to the evaluation step again after a decision is made and repeats the same steps until the design is successful.

By weighing the above characteristics, it is obvious that a mechanical design calls for an engineer's intuition and decision making ability. The artificial intelligence (AI) methods are useful in emulating those non-generalizable design steps. As it has been pointed out in Chapter 1, developing a mechanical design automation system requires more than one design methodology since each step in Figure 2.1 identifies different characteristics. Thus, several different application fields of AI were investigated in order to find the most suitable one for each design
step, namely, an artificial neural network for an initial design step and a rule-based expert system for a redesign step. At this point, it is necessary to emphasize that these AI paradigms are not the only possible choice and alternate ones can be introduced as needed.

2.2 Initial Design with Artificial Neural Networks

Artificial neural networks are composed of highly interconnected layers which attempt to achieve human neuron-like performance [22]. It is designed to emulate the human neural activities, exhibiting abilities such as learning, generalization, abstraction, classification, and recognition, all using mathematical implementation. The artificial neural networks must be trained from existing data or knowledge patterns and consists of input with corresponding target output. Once it is furnished with training knowledge, it can estimate the output when an unknown input is encountered. Training artificial neural networks using knowledge patterns can be compared with training unexperienced engineers using the knowledge of their field. Accordingly, the quality of the training knowledge influences the quality of the estimation made by the artificial neural networks. Detailed mathematical descriptions will be omitted here and can be found in references [20] and [21].

The typical way of estimating an initial design parameter is to look up existing design data with related values to locate the relevant range, then either take the closest one or interpolate within the range. When collected design data is
adequate and known to be accurate, they can be generalized and classified. As a result, an engineer may utilize readily available and well organized design data for future designs. In this aspect, the artificial neural networks are beneficial and their behavior is applicable to the initial design task. However, the way of performing initial design may vary from one component to others and from a single component to an assembled machine. Furthermore, one engineer may use different knowledge from others even for the same component design. In many cases, more than one kind of knowledge must be integrated to determine all necessary initial design parameters. Therefore, careful attention must be employed in selecting the types of artificial neural networks for a particular design situation. It is non-trivial and highly dependent on the distribution of the design data within the pattern space. It was also found that multi-level networks were required in some cases.

Four types of design data patterns are plotted in Figure 2.2. The model (A) is the ideal data pattern where all the data in one group forms a single cluster. On the other hand, the model (B) is similar to (A) but more than one cluster may be formed from one group. These types of data patterns can be found where an engineer searches for the closest value. In the model (C), data are arranged along curves (possibly lines) which indicates there exists mathematical relationship between the data in that group. The last model (D) in Figure 2.2 is a mixed pattern where the data in one group are scattered in the pattern space but resembles the model (C). When an engineer estimates a value within the range, the data pattern is normally similar to (C) or (D). If data are somewhat randomly scattered, they must
be transformed and remapped onto one of the sample patterns in order to apply the artificial neural networks.

After investigating several algorithms of the artificial neural networks, two of them are found to be especially suitable for the initial design of gears. These are Learning Vector Quantization (LVQ) [22] and Generalized Delta Rule (GDR) [20] [23]. The first algorithm works well for finding the closest value through data searching while the second one works well for estimating a value located in a certain data range. These two algorithms are illustrated in Figure 2.3 where the top one shows clusters in the pattern space of an LVQ algorithm and the bottom one
Figure 2.3 LVQ and GDR
indicates the network construction of a GDR algorithm.

An LVQ is known as a clustering algorithm which classifies available knowledge patterns by inspecting their pattern label. It then forms clusters of identically labeled patterns located within the limiting radius by measuring the Euclidean distance. The algorithm remembers the weight centers of each cluster and their pattern labels which then are used to classify new patterns, not encountered previously. Therefore, the output of an LVQ is the pattern label of a cluster to which a new pattern belongs. On the other hand, a GDR is a typical multi-layered artificial neural network constructed with an input layer, a hidden (or middle) layer, and an output layer. A network may have a number of hidden layers. However, in practice, only one or two hidden layers are sufficient for most applications [24]. Each layer has a number of nodes which are interconnected to nodes in neighboring layers. The knowledge patterns for GDR consists of inputs and target outputs. These patterns are supplied to the network in a feedforward manner to find a connecting weight matrix for each connected node pair and then those weights are adjusted through the backpropagation of errors to reduce the total system error. Once a GDR is trained, it will provide outputs for a new input pattern in a single feedforward procedure using the remembered connecting weight matrix.

In reference [28], a single LVQ network is connected to multiple GDR networks to emulate the exact steps of performing a real world design task such as a gear initial design. A similar implementation, based on object-oriented
programming, will be explained in Chapter 5. Only two artificial neural networks are briefly described here as examples, but others not covered can be applied to different strategies used in the initial design task. Furthermore, depending on the features of performing the initial design, other AI techniques such as a rule-based system, fuzzy logic system, or even a hybrids of these can be considered.

2.3 Redesign with Knowledge-Based System

Rule-based systems are widely accepted and applied in various fields. They are also known as knowledge-based systems or expert systems. They have been found to be effective in symbolic processing. Symbolic processing is one of the key features of this AI research area because it is more akin to the human thought process and uses symbols in an open-ended and flexible way to represent knowledge [5]. A knowledge-based system is judged based on how well the knowledge of the domain experts has been extracted and represented. Accordingly, the system performance is solely dependent on the knowledge embedded in it. Among the many different approaches in knowledge representation, the most active technique is that of the inference engine, which seeks a goal by an inference examination of domain specific knowledge. A typical architecture is illustrated in Figure 2.4, where three major components can be found [31][37].

In the figure, data-memory serves as a global storage which holds a goal, given facts, and presently available derived facts while rule-memory serves as a knowledge storage which holds rules. The inference engine is a rule executor
Figure 2.4 Components of Inference Engine

which accesses data-memory to match the fact (or condition) of a selected rule to the available facts. When a match is found, the rule is fired and the new fact (or assertion) derived by that rule is added to the data-memory. There are two inference mechanisms, forward-chaining inference and backward-chaining inference. These differ by their search direction. A forward-chaining system starts from what is initially known (given facts), then uses the currently available facts to make a chain of inferences until either a goal is reached or the search is exhausted. It is most suitable when there are many equally acceptable goals and a single initial fact. By contrast, a backward-chaining system works in the opposite direction. It is most appropriate for tasks such as diagnosis, in which there is a
single goal state and many potentially relevant initial information states [31]. In this aspect, a forward-chaining inference engine is selected for the redesign step. The initial state of redesign in a mechanical design is usually the fact rating is not acceptable and the number of acceptable goals are as many as the number of changeable parameters and their change intervals.

As described earlier in this chapter, the redesign step is normally achieved by applying an engineer's design experience which has been remembered in the form of conditional and decisional rules (decision making knowledge). An engineer builds this rule-like heuristic knowledge by experiencing similar design situations, usually more than once, and refining them by analyzing their similarities. This learning mechanism of humans, how humans retain and refine observed information, is not yet understood. Therefore, in a knowledge-based system, knowledge must be presented explicitly in the form of "If X is true, then Y is also true". One possible approach is to construct a hybrid system of artificial neural networks and knowledge-based systems. By patternizing rules, i.e. turning rules into patterns, a clustering artificial neural network can be applied. Similar rules can be mapped onto common clusters and a refined state of knowledge attained. This approach is beyond the scope of this investigation, but can be experimented with in the future.

The rules of mechanical design requires both symbolic and numeric data processing. They should be able to hold variable values which remain constant as the rule is evaluated and as the process continues. By weighing the characteristics
of the redesign, three types of rules are considered. The first is a condition-rule which evaluates conditions and derives new facts. Its form is;

If $X$ is true, then evaluate condition.

If condition is true, then $Y_1$ is true else $Y_2$ is true.

One example might be;

If *aspect ratio reached limit* is true,

then evaluate *pinion hardness* $<$ *hardness limit*.

If this numeric condition is true,

then *hardness is under limit* is true

else *hardness reached limit* is true.

where *aspect ratio reached limit* is the fact condition and *hardness is under limit* and *hardness reached limit* are new facts. Second, is a decision-rule which decides which action (also a new fact) should be taken such as *increase center distance* or *change gear type* in the gear design case. Its form is;

If $X$ is true, then $Y$ is also true.

The last one is a goal-rule which executes the action by changing one or more selected parameters. Its form is;

If $X$ is true, execute the action.

This modified approach is somewhat different from the pure forward-chaining inference engine, but worked well for mechanical design problem solving. Further implementation will be given in Chapter 5.
Chapter 3

Object-Oriented Approach

3.1 Object-Oriented Programming

Object-oriented programming is simply another way of programming which seems to be different in its structure from conventional programming used in procedural languages such as Pascal, BASIC, and C. Two major groups of object-oriented languages have emerged during the last decade. One group includes Smalltalk, one of the first object-oriented languages, and Actor and Eiffel which evolved from Smalltalk. They are called the pure object-oriented languages, where almost everything is an object. The other group is the hybrid structure group, whose object-oriented constructs are added to those of their predecessors, the procedural languages. The members of this group include: C++ and Objective-C derived from C; Flavors, LOOPS, and Common Lisp Object System (CLOS) derived from LISP; and object-oriented Pascal derived from Pascal [4]. Nevertheless, at the present time, the software development seems to be rapidly moving toward the object-oriented environment. The descriptions of object-oriented programming given in this section are based on references [2] through [8].

The basic elements of an object-oriented program are objects containing methods and data as illustrated in Figure 3.1. A conventional program consists of procedures and data while an object-oriented program consists only of objects
which encapsulate procedures and data. Procedures, or simply pieces of code, are referred to methods in an object-oriented program. A method is associated with a particular object and implement the unique services provided by that object. Each object acts within its own set of methods when it processes requests for service from other objects. It also sends messages to other objects to ask for assistance in the completion of its tasks. An object is often called a black box since its methods and data are encapsulated. In other words, a message sender object does not know how a message receiver object will carry out a task, yet a message receiver knows exactly what to do. Message passing is the only means of communicating between objects, importing data for local manipulation, and accessing data encapsulated in receiver objects. This information hiding or encapsulation ability is one of main features of object-oriented programming.
Other terms used in object-oriented programming are class, superclass, subclass, and instance which refer to an object in various contexts. Many different objects may behave in very similar ways or even exactly the same way in providing certain services. The common methods and data that implement these services can be abstracted in one place, called a class. Data abstraction is another feature in object-oriented programming. A class may have multiple subclasses (children or derived classes) with which it forms a class hierarchy. Figure 3.2 illustrates this hierarchy with an example of "Professional Occupation" classes. In subclasses, more specific common factors are summarized such that "Mechanical Engineer" is a more specific occupation than "Engineer". In contrast, more general common factors are abstracted higher in the hierarchy. These higher classes are referred to as superclasses (parent or base classes), like the "Professional Occupation" class in the figure. By organizing classes in a hierarchical form, object-oriented programming offers a powerful inheritance mechanism which can not be found in any procedural languages. Methods and data in a superclass are automatically inherited by its subclasses. Therefore, common methods need to be written only once in a superclass, resulting in simplified implementation of subclasses. Furthermore, those methods can be overwitten or expanded in subclasses when modification of behavior becomes necessary, as shown in Figure 3.3. Some languages, like Smalltalk, can have only one superclass while other languages, such as C++ and CLOS, may have more than one superclass. With this advantage, expanding and updating software is simply a matter of adding a new subclass with
Figure 3.2 Example of Class Hierarchy and Instances of its classes

Figure 3.3 Overwriting or Expanding Superclass Methods
Figure 3.4 Expanding Class with Subclasses

new or modified methods. Figure 3.4 shows how the "Mechanical Engineer" class can be expanded with new subclasses. This unique inheritance mechanism makes object-oriented programming more attractive to the programmers of complex and ever-changing applications.

In Figure 3.2, "Tom", "Jane", and "Bill" are individual engineers. "Tom" and "Jane" are called the instances of the class "Mechanical Engineer" while "Bill" is called the instance of the class "Electrical Engineer" in terms of an object-oriented programming. Each engineer may have different characteristics and abilities to perform a task such that "Tom" has a college education and knows how to design mechanical components while "Jane" has high school education and knows how to use commercial CAD software. When an instance is created at run-time, passed information (data from a message sender object) is localized to instance variables (data in a message receiver object) and all methods of its
originator class including ones from its superclasses become available. This mechanism is represented in Figure 3.5. Again referring to Figure 3.2, once "Tom" is hired as a mechanical engineer (compared with creating an instance) he is ready to receive a work order. His manager does not care how he performs the work, but expects a result from him. His abilities or knowledge is hidden within him like a black box, yet he knows exactly how to perform the work. If "Tom" does not know how to do the work, then he needs to be trained with new knowledge (new methods) or he may be replaced by another engineer (other
compatible object). In object-oriented programming, new methods can be added and an object itself can be replaced by a new one without affecting the other methods or objects in a program. This is not true in conventional (or procedural) languages, where the entire structure may be affected by a small change in a program. The object-oriented way of thinking seems more natural to a human than the traditional procedural approach.

Polymorphism is another feature of object-oriented programming. As already mentioned, objects act in response to the messages they receive. Different objects may receive the same message, but may act in completely different ways. Let's assume that a manager sent a work order (message) to both "Tom" and "Jane". The work order may be "Read this requirement and finish the work by tomorrow". When "Tom" receives this order, the result will be a mechanical component design such as a gear set or a shaft since mechanical component design is his specialty (specific methods). Meanwhile, "Jane" will finish a work drawing using a commercial CAD package. They act differently on the same message because data they received are different and methods they know are different. This phenomenon is referred to as polymorphism which is also a unique feature of object-oriented programming.

After all, objects are even viewed as software ICs in reference [7]. When an electrical engineer designs a circuit board, he or she only cares about the behavior of the selected ICs and does not care of their internal circuits and layers. If the selected ICs do not function as expected, any of them can be replaced by
different ones without redesigning the entire circuit board. With this view point, objects can be compared with hardware ICs. Many features of object-oriented programming have been described, which are more like common features of languages in this category. However, depending on languages, even more features are available. Dynamic binding, visual programming, persistence, and binary large object (Blob) are some features which can be found in other object-oriented languages. With these new techniques, developing mechanical design automation systems based on object-oriented programming techniques might prove to be beneficial.

3.2 Object World of Mechanical Design

In Figure 3.4, the expanded class hierarchy of a "Professional Occupation" is illustrated. Three new subclasses, "Engineering Manager", "Design Engineer", and "Manufacturing Engineer", have been added to the class "Mechanical Engineer". Three individual engineers are also added as shown in Figure 3.6, which represent the instances of each of three new subclasses. Let's assume that the Figure 3.6 represents the object world of a gear manufacturing company. "Mike", the manager, knows how to play the role of a manager. "Tom" knows how to design gears based on design specifications and "Bill" knows how to manufacture them using blue prints. The manager sends a work order "Do this work" (message) with user requirements (data) to the design engineer, "Tom". No matter how "Tom" designs a gear set, he will complete the work and return a blue
Figure 3.6 Object World of Gear Manufacturing Company

print to the manager. Then, the manager sends a work order "Do this work" with blue prints to the manufacturing engineer, "Bill". He will also complete the work and return a set of gears. This might be one example of how the work flows at a gear manufacturing company.

Let's get down to even smaller world of objects at the gear manufacturing company example. "Tom" has his own design team where his role is that of a supervisor. The design team is formed with design engineers, assistants, and a
Figure 3.7 Object World of Gear Design Team
drafter. Referring to Figure 3.7, as soon as "Tom" inspects the design specifications, he requests an initial design of gears from the engineer "Joe" who has many years of design experience and knows how to perform an initial design. "Joe" may request help to his assistant "Susan" to look for past designs if any similar designs exist. Once the initial design is done, "Tom" will request the assistants "Nancy", "Mark", and "Jack" to find the material properties, to calculate the gear geometries, and to calculate the AGMA power rating respectively. If the design rating fails, "Tom" sends all three assistants to the engineer "Ron" who knows how to analyze the design rating results and how to select and change necessary parameters. They get together and have a redesign meeting. "Ron" will query each assistant in order to analyze the reason for the design failure and make a redesign decision. Then, "Ron" will send a redesign result to "Tom". He repeatedly requests the same work of the assistants in the design evaluation team and "Ron" until the design succeeds. After the design is done, he will request that engineer "Robert" design a gear blank and the drafter "Jane" generate blue prints. This will conclude all design procedures which "Tom" should supervise. At last, "Tom" will send out the blue prints and any extra design outputs to the manager.

The above scenario is already described in Chapter 2 with Figure 2.1. In Chapter 2, the design steps were explained in a procedural manner while the steps described above are based on an object-oriented concepts. In real world practice, one engineer may handle the entire design process with the aid of well organized past design data, commercial software, and computers. However, by assuming that
those resources are not available or incomplete, the model in Figure 3.7 becomes valid. With this object-oriented model of a gear design team example, the mechanical design steps can be reconstructed as illustrated in Figure 3.8 where all objects are the instances of their respective classes. This figure also represents the structural foundation of an Artificial Mechanical Designer which will be discussed in the following chapter. Each small component of the design process is made into an object. Some of them have heavy responsibility and others play a smaller role in

![Object-Oriented Model of Mechanical Design](image)

**Figure 3.8** Object-Oriented Model of Mechanical Design
the portrayed design process. Each object in the model of the mechanical design in the figure is identified as a task-object (i.e., a task-performing object). A task-object defines the general behavior of its instances, as will be explored in the following section. Each instance knows exactly how to carry out a requested task just like the human engineers or assistants in the gear design team example. By using object-oriented programming, the real world design simulation becomes more natural and realistic.

3.3 Definition of Task-Object

One of the main objectives in this project is to determine how individual design automation systems should be developed to be compatible with each other in terms of communication. One possible approach is to specify the communication protocols, namely receiving and requesting a task, and implement them at higher

![Diagram](image_url)

**Figure 3.9 Object vs. Task-Object**
level of each individual design system. As a result, each design system will know how to talk to other design system. A task-object was primarily designed to achieve this objective. A task-object is an object which has been renamed for use in engineering design applications. In addition to that, other terms in object-oriented programming have been renamed. The comparisons of the original names in an object and new names in a task-object are given in Figure 3.9. A task-object becomes a task requestor object when it sends a task-order and it becomes a task performer object when it receives a task-order.

The abstract class TaskObject (using Smalltalk convention: class names are capitalized and italic-bold faced and method names are just italic faced) has been defined to represent the general behavior of a task-object. The elements of the class TaskObject and its basic mechanisms are described in Figure 3.10. Referring to this figure, there are two instance variables, return-parameter-set and recorder, which are common to all subclasses of the TaskObject class. First variable return-parameter-set is a dictionary (in Smalltalk convention) which is initially empty and will hold the selected local parameters. These local parameters will be returned when a task requestor object sends a task-order, like "show return parameter set". The second variable recorder contains an instance of the class Recorder which knows how to record certain messages or values and where to record them. The class TaskObject has several general methods embedded in it, such as initialize, error-handle, perform, task-perform, task-request, etc.. Each subclass will have its own task specific methods as well as overwritten methods.
Figure 3.10 Elements of TaskObject Class and its Basic Mechanism
The two methods, arguments-inspect and task-perform in the class TaskObject are default methods and they are expected to be overwritten by its subclasses. If they are not found in a subclass, the default methods will be invoked and cause a fatal error. As a result, a task will be terminated and an error message will be sent to a task requestor object. In addition to them, local-parameter-access methods should be added by subclasses as needed. These local parameters are not to be confused with an instance variable return-parameter-set. They represent specific parameters defined in subclasses and only those parameters expected to be returned will join the return-parameter-set.

An instance of the class TaskObject, a task performer object, is created by receiving a task-order "initialize" with a set of arguments from the outside world, a task requestor object. After initializing instance variables (including local parameters), it receives the second task-order "perform" which is an actual task request order. By receiving this order, the perform method first inspects the arguments sent through the arguments-inspect method, then passes the control to the task-perform method. This method accesses other specific methods (defined in subclasses) in their orders. If any necessary arguments are missing or invalid or any error occurs while performing, the control is returned back to the perform method, where the task is terminated with an error message through the error-handle method. While performing a task using the specific methods of one task-object, they may ask for assistance from other task-objects. In the task-request method, the way to request a task to another task-object is defined. When a
Figure 3.11 Task Performing Mechanism
Figure 3.12 Task Requesting Mechanism
performer object completes a task, it will send a "succeeded" message to its requestor object. Of course, it will send a "failed" message if any error occurred. A requestor object, then, will send another task-order, such as "show return parameter set" to get the task result. Or, if a requestor object is only interested in a particular parameter, it will access the respective local-parameter-access methods to get the value. The mechanisms of the task-perform method and task-request method are illustrated in Figure 3.11 and 3.12 respectively.

As a summary, the class TaskObject knows how to receive a task-order, how to control a task flow, how to send a task-order, and how to access other task-objects. It also has methods that know how to store parameters, how to record messages and values in the designated place, and how to handle errors. However, it does not know how to perform an actual task since all task specific methods should be defined in its subclasses. It just abstracts the general behavior and serves as a template.

3.4 Developing Subclasses of TaskObject Class

All task-objects in mechanical design perform their unique tasks, which implies that more specific task-objects need to be defined as subclasses of the class TaskObject. In the gear design team example, two groups of engineers whose roles are significantly different were introduced, namely an engineer group and an assistant group. To represent these two groups, new subclasses are added to the class TaskObject. They are the TaskPerformer and the TaskAssistant classes.
Figure 3.13 A Base Class Hierarchy of Task-Objects
Furthermore, the TaskPerformer class is expanded with more specific task subclasses, namely, the ControlPerformer and the MethodPerformer classes as illustrated in Figure 3.13. The class ControlPerformer represents the group of experienced engineers whose roles are controlling design strategies. The members of this group include a design supervisor, an initial design engineer, a redesign engineer, and an optional design engineer. Four subclasses are defined to represent these engineers, and they are the DesignController, InitialDesigner, ReDesigner, and OptionalDesigner in the figure. Engineers need particular design knowledge and design method to carry out a task. Different design methods are usually used for different types of tasks. Furthermore, engineers will choose their favorite method which is determined by their unique design experience. Therefore, it is necessary to organize those design methods into a separated class MethodPerformer. Any specialized design methods, including various Artificial Intelligence techniques, belong to this group. Objects in the MethodPerformer do not control the design in any manner, but only assist objects in the ControlPerformer by performing a certain design method. In fact, their tasks are completely separated from the mechanical design tasks. Meanwhile, the class TaskAssistant represents the group of assistants, such as "Nancy", "Mark", and "Jack" in the gear design team example (Figure 3.7). Three subclasses, MaterialProperty, Geometry, and Rating, are added to emulate these tasks. The tasks assigned to these assistants do not usually require specific design knowledge.
Any tasks which use formalized knowledge, such as standardized formulas, tables, etc., can be organized under this class.

The classes shown in the figure are all abstract classes except the class DesignController. An abstract class defines only general behavior and can not have an instance of itself. Therefore, these abstract classes must have task specific subclasses to perform a real design task. Developing a mechanical design automation system with task-objects can be viewed as developing proper subclasses which will together emulate a desired mechanical design task. In the case of the DesignController class, general mechanical design procedures are already predefined based on the model of mechanical design (Figure 2.1 and Figure 3.8). Therefore, this class can have an instance of itself. If, however, a different design procedure is desired, a new subclass should be added.

As an example, possible class hierarchies for a gear design automation system are illustrated in Figure 3.14 and Figure 3.15. All task-objects in the classes TaskAssistant and MethodPerformer are assumed to be available to an engineer. They are not intended to be developed by an engineer. By assuming they are all available either commercially or non-commercially, an engineer only needs to obtain (or purchase) his or her choices of task-objects and is not burdened by having to develop them. Or, if it is preferred, an engineer may also develop them. Meanwhile, the subclasses of ControlPerformer, such as GearInitialDesigner, GearReDesigner, and GearBlankDesigner in Figure 3.14, are supposed to be developed by an engineer since these task-objects will control design strategies.
Figure 3.14  TaskPerformer Class Hierarchy Example for Gear Design
Figure 3.15 TaskAssistant Class Hierarchy Example for Gear Design
This is an ideal case where an engineer should know how to program and develop a task-object. Can an engineer avoid development of these task-objects as well? The answer is yes only if there exists predefined subclasses which are applicable to the system being developed. If so, an engineer only needs to organize design data and knowledge and make them available to the task-objects.

If all tasks in the real world design practice can be converted into task-objects, big or small, important or trivial, developing a design automation system becomes as easy as playing with children's toy blocks. Engineers should be able to put task-objects together to build their own design automation system using their own knowledge without having extensive computer expertise and other relevant skills. Task-objects fit together since they are designed that way. They are interchangeable if they know how to perform the same specific task, perhaps by using different methods. They are expandable and modifiable by adding new subclasses with expanded or overwritten methods. They can be reused in the development of future design systems. By utilizing the concept of the task-object, all these advantages can be obtained with minimal effort.
Chapter 4
Implementations of Artificial Mechanical Designer

4.1 Definition of Artificial Mechanical Designer

An Artificial Mechanical Designer (AMD) is an instance of the class DesignController which is a subclass of the ControlPerformer and TaskObject classes introduced in Chapter 3. It represents the design team which is able to perform a mechanical component design. Its structural foundation came from the object-oriented model of the mechanical design process (Figure 3.8). In Figure 4.1, a single level AMD construction is illustrated. Although an AMD is just an instance of the class DesignController, the term AMD will be used to refer to the entire structure in the figure since all task-objects connected to it must cooperate in performing a single component design task.

In Chapter 1, it was mentioned that an AMD is also a task-object which shares the behavior and communication protocol of the task-object, but internally is more complicated. Therefore, AMD will work just like any other task-objects as defined in Chapter 3. It receives a task-order from other AMDs or task-objects and performs its unique task. It also knows how to request a task from other AMDs or task-objects and how to access their local parameters. The only difference between AMD and a task-object is that AMD is able to design a particular mechanical component while a task-object is not. A task-object performs a task which is only a small portion of a complete mechanical component design.
Figure 4.1 Structural Model of Artificial Mechanical Designer
The class *DesignController* contains the *task-perform* method in which the most common procedures of a mechanical component design are programmed. This method is the heart of AMD and knows how to control the design steps. AMD begins its task by receiving a task-order "perform". As shown in Figure 4.2, the first step is to inspect the passed arguments with the *arguments-inspect* method. The passed arguments are user inputs in this case. If arguments are missing or invalid, the *arguments-inspect* method creates an instance of *InputReceiver* and directs it to acquire input values from the user. In a general task-object, this is not true. An error message is produced if arguments are missing or invalid. Automatic requesting of input values from the user is a unique feature of the class *DesignController*.

Once the passed arguments are inspected, the *task-perform* method is called. It requests an initial design task from an instance of *InitialDesigner*. After getting a set of initial design parameters, it sends task-orders to instances of *MaterialProperty* and *Geometry*. Then, it requests the design rating task from an instance of *Rating* by providing the *MaterialProperty* and *Geometry* instances. The *Rating* object can access those two task-objects to obtain values in order to determine a rating result. If the rating result turns out to be "failed", it sends a redesign task-order to an instance of *ReDesigner*. This time, the *MaterialProperty*, *Geometry*, and *Rating*, are sent to the *ReDesigner*. The *ReDesigner* directly accesses local parameters of them and make necessary changes to the selected parameters after a redesign decision is made. Consequently, the *task-perform*
Figure 4.2 Procedures in *task-performer* Method of *DesignController* Class
method does not need to recognize the redesign decision. It just requests the same
task-orders from all three design evaluators and the redesigner until the result of
the design rating is acceptable. Finally, it requests any optional design task from
the OptionalDesigner. After these sub-tasks are successfully completed, the design
output can be obtained by accessing the variable return-parameter-set and
appropriate local parameters.

The InitialDesigner, ReDesigner, and OptionalDesigner objects are
programmed to access one or more subclasses of MethodPerformer based on design
methodologies chosen by the engineer. Ideally each of these three task-objects
should be able to use any method chosen. However, in this implementation, one
type of design method has been pre-selected for each one of them. The reasons for
this approach will be given later in Section 4.4. The purpose of this chapter is to
present an implementation example of AMD. It is not intended to provide an
idealized AMD development. Also, the description of the class OptionalDesigner
is omitted because its implementation is similar to other ControlPerformers.
Therefore, the implementation of the AMD discussed throughout the remaining of
the chapter will serve as only one possible development model. The procedures
built into the class DesignController are perhaps the most common ones. If any
modification is necessary, a new subclass can be derived from this class. This can
be done by providing a new task-perform method that performs the desired design
steps and related methods (only specific methods). The structural model of an
AMD shown in Figure 4.1 will change if modifications are made, because its task
specific methods have been changed. The AMD is not a prefixed design model, as its structural shape is dictated by the specific design task that is being emulated.

4.2 Subclasses of TaskAssistant Class

There are three primary subclasses in this category as previously shown in Figure 3.13. They are MaterialProperty, Geometry, and Rating classes. Each of them has one or more subclasses. The TaskAssistant class has one local parameter, change-flag, and one method, initialize-local-parameter, which will set change-flag to "false" initially. This class is defined only for the purpose of categorizing the assistant task-objects. Task-objects that use formalized knowledge can be organized under this class. The class MaterialProperty will be discussed last since it has some unique features which are different from other TaskAssistant objects.

In Figure 4.4, the implementation of the Geometry class and its subclass are illustrated. The Geometry class is also defined for categorizing purposes and has only one method, group-title, which is used to return the title when the list of design parameters is printed out. Three methods found in its subclass are commonly required: initialize, arguments-inspect, and task-perform methods. The method initialize contains the procedures for initializing local parameters in a subclass if any exists. The method task-perform should implement the basic parameter calculation procedures based on the passed arguments. A specific method defined in the subclass contains calculational procedures for each design parameter. These specific methods are also used to access a value calculated by
Figure 4.3 Implementation of Geometry Class

Figure 4.4 Specific Method Example in Subclasses of Geometry Class
itself, similar to the *local-parameter-access* methods. Specific methods of this type are utilized whenever a number of local parameters need to be reduced. When an equation requires other values to be calculated, which is the usual case, it will get the values from the appropriate specific methods.

One example of the relationship between specific methods is represented in Figure 4.4. When any of the basic design parameters need to be changed for redesign, an instance of the *ReDesigner* class will access the appropriate *local-parameter-change* methods. These methods should know how to change other related basic parameters as well. The basic design parameters are usually defined as local parameters in a subclass and their initial values are set to the ones sent from a task requestor object as arguments.

The last method in a subclass is the *return-parameter-set-access* method. It is an overwritten method to the general one defined in the *TaskObject* class. Since subclasses of *TaskAssistant* do not store all return values as local parameters, it is necessary to trigger the specific methods to calculate un-stored values. This method gets required values, stores them in a Dictionary (a collection object in Smalltalk convention), and returns the Dictionary to a task requestor object.

In Figure 4.5, the implementations of the *Rating* class and its subclass are illustrated. The *Rating* class has two local parameters, *rating-log* and *rating-point*. The *rating-log* contains a boolean expression which is the result of the most recent design rating while the *rating-point* contains a value which represents the ratio of the applicable value over the required value. These values will be determined by
Figure 4.5 Implementation of Rating Class

the task-perform method of the subclass. Two local-parameter-access methods are defined for above two parameters. The behavior of the methods defined in its subclass are similar to the ones in a subclass of the class Geometry. The differences are that the task-perform method has the design rating procedures instead and there
are no local-parameter-change methods. The specific methods in a subclass will access subclasses of Geometry and MaterialProperty to obtain the values required to carry out the design rating.

In Figure 4.6, the implementation of the MaterialProperty class and its subclass are illustrated. The MaterialProperty class has a unique method which other task-objects do not have. The method known-property contains the constant values of all known properties, such as modulus of elasticity and Poisson's ratio. When a task requestor object only needs a known property, it will directly access it via this method instead of the initialize method. Another difference is that the task-perform method does not have any specific task procedures and always returns "true" (in Smalltalk convention). It is defined as a dummy method to prevent the task from terminating with an error message. The arguments-inspect method is defined in the MaterialProperty class instead of its subclasses since the arguments are common to all subclasses. The required arguments for this class consists of the component-name, grade, and hardness. The local-parameter-access methods for these parameters, as well as the known properties, are predefined in the class MaterialProperty. All other changeable properties must have their own specific methods, such as allowable-bending-stress, allowable-contact-stress, and hardening-method. The return-parameter-set-access methods are defined in both the MaterialProperty class and its subclass. One in a subclass is an expanded method. The local-parameter-change methods are also predefined in the MaterialProperty class. When a parameter is changed, no further action will be
Figure 4.6 Implementation of MaterialProperty Class
taken. However, when a property is requested from a task requestor object, a new value will be calculated based on the changed parameter. This differs from the behavior of subclasses of the *Geometry* class.

Although there exists some implementational variations, all subclasses of the *TaskAssistant* class are developed based on the requirements of the task-object. It is necessary to emphasize that the majority of the specific methods in the subclasses play two roles. The first role is that of performing its task, namely calculating a design parameter or related value. The second role is to return a value. Therefore, the *local-parameter-access* methods are not defined in any subclasses except those of the *MaterialProperty* class. They are needed to return known properties (non-changeable properties) in this class. Based on Figure 3.15, several subclasses are developed for the gear design application which will be introduced in Chapter 5. They are the subclass of *MaterialProperty: Steel*, the subclasses of *Geometry: AGMA20102, AGMA908B89,* and *GearParallelExternal,* and the subclass of *Rating: AGMA2001B88.* AGMA is the abbreviation of American Gear Manufacturers Association and the symbols after AGMA indicate their respective standard numbers. The desing and organization of a class hierarchy is the choice of the developers. The example of a gear geometry class hierarchy is introduced in Figure 4.7 as one possible approach. However, more efficient class hierarchy may reduce programming overhead.
1: Non-Parallel and Non-Intersecting
2: These classes have two superclasses

Figure 4.7 Example of Gear-Geometry Class Hierarchy
4.3 Subclasses of MethodPerformer Class

One abstract class, four method subclasses, and related objects are developed for the MethodPerformer class. As previously mentioned, the tasks performed by its subclasses have no link to any specific mechanical design strategies. A MethodPerformer should be designed to perform as a generic methodology which can be used for any applicable design step. Currently implemented class hierarchies are illustrated in Figure 4.8, where the gray colored subclasses contain the actual algorithms or procedures. Two other classes, NetObject and ExpertObject, which are not subclasses of TaskObject class are also defined. Subclasses of the NetObject are the necessary elements of Artificial Neural Networks while subclasses of the ExpertObject are required for rule-based systems, such as a forward-chaining inference engine.

The classes LVQNet, GDRNet, and LVQwithMultiGDRNet are the applications of the Artificial Neural Networks. The LVQNet and GDRNet are organized under the class SingleNeuralNet. These task-objects have their respective Artificial Neural Network algorithms while the LVQwithMultiGDRNet class knows how to construct multi level networks with the LVQNet and GDRNet task-objects. Accordingly, the characteristics of these two groups are obviously different and for this reason they are separated in the class hierarchy. If more multi-level network task-objects are desired, a new abstract class, such as MultiNeuralNet, can be defined to organize them.
Figure 4.8 Current Implementation of MethodPerformer Class Hierarchy
In Figure 4.9, the general procedures of training and consulting Artificial Neural Networks are illustrated. These have already been discussed in Chapter 2. A pattern file for the training contains existing design data, input patterns and target output pairs. Once a network is trained, it will generate a net file in which clustering results or connecting weight vectors are stored. The net file used in this implementation has a header section prior to the above data where other information is stored, such as input list, output list, and a flag (boolean expression) that indicates whether pattern transformation is required or not. Such information is supplied by a user through a setup process. Specially, input and output lists contain actual names of design parameters. Therefore, after a training is done, the header section in a net file becomes the link to a certain mechanical design task. The
training process is normally a time consuming task due to the substantial number of training iterations and is more complex than the consulting process due to the trial and error in setting up the network parameters. In comparison, the consulting process is done in one iteration and does not require any parameter setup. When a new input is encountered by a fully trained network, an estimated output will be provided based on the information stored in a net file.

In Figure 4.10, the element objects of the *LVQNet* and *GDRNet* classes are illustrated. The members in the *LVQNet* are the *NetSpace*, *NetCluster*, and *NetPoint* objects. The *NetSpace* object has the list of the *NetCluster* objects and knows how to form clusters with input patterns which are *NetPoint* objects. It has a method to find the closest cluster center to a particular *NetPoint* for consulting in order to determine a pattern label for that *NetPoint*. The *NetCluster* object has a list of *NetPoint* objects, namely patterns, and its cluster center (also a *NetPoint* object). It knows how to distinguish whether a given *NetPoint* should belong to its cluster or not. It measures the distance between a candidate of a cluster center (often referred to as a neuron) and a given *NetPoint* by calculating the Euclidean distance. Then, it forms its own cluster with all *NetPoint* objects that belong to it by comparing the distance with the limiting radius and inspecting a pattern label. The *NetPoint* object is the smallest object in the class *LVQNet*. It contains the pattern values, the pattern label, and the transformed pattern values which are provided by a message sender object. There are three different pattern transformation methods defined in this object. The first method changes the pattern
Figure 4.10 Element Objects in LVQNet and GDRNet Classes
values to new values. In contrast, the next one changes it back to its original values. If it is necessary to transform again, then the last one will transform it to the existing transformed values. Unlike the first method, this method will not accept new pattern values. Transformation methods are required if existing design data (patterns) need to be transformed to a different pattern space due to an unsuitable pattern distribution.

In the case of the GDRNet class, the members of a network include NetNode, NetLayer, and Network objects [67]. The Network object represents the whole network construction and has a list of NetLayer objects. It knows how to activate NetLayer objects and how to adjust overall system error. The NetLayer object includes a list of NetNode objects. It knows how to connect itself to neighboring NetLayer objects and how to activate each NetNode object in it. The NetNode object is the smallest object in the GDRNet. It has methods to initialize threshold and weight values for training purposes. It knows how to connect to NetNode objects in the neighboring NetLayer objects and how to activate itself and those connected NetNode objects.

The implementation of the SingleNeuralNet class and its subclass are represented in Figure 4.11. Their general structures are very similar to ones previously discussed. However, the SingleNeuralNet class has a second initializing method, namely the training method. This method is used to initialize an instance for training purposes while the initialize method (a common method) is used for consulting purposes. In the figure, only the consulting mechanism is illustrated.
Figure 4.11 Implementation of *SingleNeuralNet* Class
Since all *MethodPerformers* do not recognize what particular design is being done through them, they can not organize the design parameters to be returned. A couple of general methods are defined to support this task in the *SingleNeuralNet* class. They are the *read-net-file-header* and *prepare-returns*. It was mentioned that a net file has a header section prior to the network data. This header section includes the actual names of design parameters used in training pattern files. The *read-net-file-header* method reads these names and stores them into the local parameter *expected-output*. After the task is completed, the *prepare-returns* method forms the variable *return-parameter-set* based on the contents of the variable *expected-output*. For the same reason, the *task-perform* method is defined in both the superclass and the subclass.

Another Artificial Intelligence technique implemented is a forward-chaining inference engine. The forward-chaining system requires two objects, namely the *ExpertRule* and the *ExpertFact* objects. As previously explained in Chapter 2, starting from a known fact, a forward-chaining system searches through all available rules to locate the goal fact. The action of matching conditional rules with resultant assertions is called forward-chaining. The implementation of the class *ForwardInference* is illustrated in Figure 4.12, where some methods are based on the *InferenceEngine* class found in reference [66]. This class is activated by receiving the list of *ExpertRule* objects and starts to perform reasoning upon receipt of the goal fact. The *ExpertRule* object is initialized with one or more conditional facts (or simply a condition), one or more derivable facts (or simply an action), and
Figure 4.12 Implementation of ForwardInference Class
an explanation of the action. The forward-chaining mechanism is programmed in
the task-perform method. This method "fires" one of the available ExpertRule
objects and check its conditional fact. If the conditional fact is true, then the
ExpertRule object will create an instance of an ExpertFact object and add it to the
active fact group. Every time an ExpertRule object is fired, the task-perform
(inference engine) method checks whether the goal fact has already been derived or
not through the evaluate-goal method. After the task is completed, a task requestor
object can get an explanation of action rationale, namely how it found the goal fact,
by accessing the explain-actions method. This method returns the chain of actions
taken from the initial and derived ExpertFacts to attain the goal. The
implementation discussed is perhaps the simplest inference engine possible.
However, it worked well for the gear design application as will be shown in the
following chapter.

4.4 Subclasses of ControlPerformer Class

It is purposely planned to explain the TaskAssistant and MethodPerformer
classes prior to the ControlPerformer class. The names of its subclasses are
identified by their tasks in the mechanical design procedures, such as initial design,
redesign, and optional design. These names are compatible with the ones in the
procedural model of the mechanical design steps (Figure 2.1), an ordinary flow
chart. To avoid any confusion in understanding the object-oriented model of the
mechanical design steps (Figure 3.8), the similar names are chosen in this implementation.

As an example, assume the \textit{LVQNet} is the applicable design method for the initial design step of a particular component design. Programming the initial design steps in a conventional way would be top-down procedures. In such a way, the whole sequence, i.e., control procedures and LVQ network procedures, will be considered as the initial design task-object. If the same procedures are programmed in an object-oriented way, all possible task objects would be extracted from the sequences and programmed one object after another. Then, these objects are put together to form the initial design task-object. Referencing to Figure 4.13, the only remaining task in the initial designer's black box is the control procedures which does not include the LVQ algorithm. Therefore, a more precise name of the initial designer is perhaps the "LVQ Controller". If another design method, like an inference engine, is used for the initial design, then the name could be the "Inference Engine Controller". The initial designer may have as many names as numbers of applicable design methods used in practice. The proper approach is also presented in the figure, where the initial designer task-object is an empty box that can accept any available controller objects of the design methods. However, as previously pointed out in Section 4.1, only a single type of design method is preselected for the classes \textit{InitialDesigner} and \textit{ReDesigner}. As a result, the \textit{InitialDesigner} can accept an Artificial Neural Network, a single level or multi-level, and the \textit{ReDesigner} can accept an inference engine application. If it is
Figure 4.13 Control Object vs. Method Object

preferred, the names can be replaced by the "Neural Net Controller" and the "Inference Engine Controller" respectively.

In Figure 4.14, the implementation of the class InitialDesigner and its subclass are illustrated. All methods in a subclass except specific methods are commonly required ones. However, only the net-input-pattern method will cause an error if this method is not presented in a subclass. The rest of them can be
Figure 4.14 Implementation of InitialDesigner Class
omitted if not necessary. The *task-perform* method defined in a superclass first accesses the *net-input-pattern* method in a subclass to get an input pattern. It initializes a *NetPoint* object with this input pattern. Then, it accesses the *transformed-pattern* method. If this method is presented in a subclass, it will get a transformed pattern and transforms the *NetPoint* object. And it requests a task of a selected Artificial Neural Network application. After it receives an output (the local parameter *net-output-set*), it accesses the method *net-output-adjust* if one is presented. This concludes all Artificial Neural Network procedures. Finally it accesses the method *default-initial-values* in a subclass. Any design parameters which are usually determined by default values are contained in this method. Values for such parameters can be requested of a user as a design input if it is preferred in order to prevent limiting flexibility.

The implementation of the class *ReDesigner* and its subclass are illustrated in Figure 4.15, where an inference engine is the choice of a design method. The *task-perform* method accesses three methods, the *set-material-assistant* and the *add-all-rules* defined in a subclass and the *make-decision* defined in a superclass. The add-all-rules method is required and an error will occur if it is not presented. Its default method defined in a superclass adds two rules, "rule 1" (the first condition-rule) and "rule n" (the last goal-rule). The "rule 1" accepts the given fact "evaluation failed" and will derive "redesign failed" while the "rule n" will derive the goal fact "done". Therefore, the task will be terminated due to this chaining; "evaluation failed" - "redesign failed" - "done". The method *set-material-assistant*
Figure 4.15 Implementation of ReDesigner Class
is only required when more than one material are used in a single AMD (Artificial Mechanical Designer). If this method is not presented in a subclass, the task-perform method will assume that only one material is used. The role of the make-decision method is to send a task-order to a selected inference engine application and to request the explanations of the actions taken.

To use any inference engine application, a rule-base must be predefined. Usually rules are stored in a separated rule file. In this implementation, rules are defined as specific methods in a subclass. There are three types of rules and they are condition-rule, decision-rule, and goal-rule which are distinguished by the way of acting. Their basic elements and sample rules in Smalltalk convention are represented in Figure 4.16. A rule consists of five elements, namely id, fact, action, explain, and why. The id is used to identify rules. The fact is represented symbolically and serves as a conditional fact of the rule. In other words, if this conditional fact is true, the action part of a rule will be executed. The action part of a condition-rule has the structure of "If (numeric condition) is true, then (symbolic fact 1) is true else (symbolic fact 2) is true". A condition-rule derive a new fact, such as "helix angle is under limit". In the case of a decision-rule, its structure is simply "(symbolic fact) is true". A decision-rule derives a new fact which is a redesign decision, such as "increase hardness". A goal-rule has a numeric equation in the action part. It may have some numeric condition part to select a proper equation. As shown in the figure, both a condition-rule and goal-rule directly access the local parameters of MaterialProperty, Geometry, or
Figure 4.16 Examples of Rules
*Rating* instances in order to obtain numeric values for their *action* part. Specially, a goal-rule directly accesses the *local-parameter-change* methods to change the local parameters based on a redesign decision. These rules are the connections to other *TaskAssistant* instances. Meanwhile, both the *explain* and *why* parts are text based explanations about the *action* and *fact* respectively. They will be used to explain how a certain redesign decision is made and what orders are taken when a task requestor object sends a task-order, like "explain all actions taken". An explanation example is "Helix angle is checked because material grade is 2".

All necessary task-objects in the AMD structure are described in terms of their local parameters and methods. They seem to be very complicated in their structure. In fact, there exists some programming overhead to utilize the concept of the task-object. However, they all share common behavior defined in the class *TaskObject*. As a result, they are all compatible in terms of receiving a task-order and requesting a task. Together they construct a single AMD and work together to complete a mechanical component design. Furthermore, each of them is reusable to build another AMD without any changes. When a particular component design AMD is developed, an engineer, as a design system developer, will only need to develop the subclasses of the *ControlPerformer* class, such as *InitialDesigner* and *ReDesigner*. Their knowledge is represented as a design data pattern file for Artificial Neural Network task-objects and as a rule-base for inference engine task-objects.
4.5 Developing Multi-Level AMD Construction

A task-object has the ability to communicate to other task-objects and knows how to work with other task-objects as a part of an AMD, which is also a task-object, but more complex than a single task-object. It behaves exactly the same way that a single task-object does. Therefore, a more complex AMD can be constructed with multiple AMDs, even with a mixture of AMDs and task-objects. As mentioned at the beginning of this chapter, the AMD is, in fact, an instance of the class DesignController which is of course a task-object. This is how the AMD can behave just like a single task-object. Therefore, communicating between two AMDs can be considered as communicating between two DesignController task-objects. It is also true that any task-object in the AMD can request a task from a single task-object or other AMD. By contrast, all other task-objects in the AMD except the DesignController can not receive a task-order from others. Only the DesignController can receive a task-order. This is illustrated in Figure 4.17, where the possible task requesting directions between AMDs and task-objects are shown with arrows.

Developing an AMD that consists of other AMDs and task-objects requires the same procedures as developing a single AMD. However, it is obvious that design strategies in the class DesignController and design methodologies used in the subclasses of the ControlPerformer are different from the ones in a single AMD. In other words, designing a multi-component unit or a assembled machine is not like designing a single component. Especially, the design evaluation steps now
require an engineer's decision making. Evaluating lists should contain if any interference exists between components and if each component is properly designed to deliver the requirements of the entire unit or machine. As an example, if a Gear-Drive-System AMD is being developed with multiple AMDs, such as a Gear AMD, a Shaft AMD, and a Key AMD, it will be necessary to check if the gear set fits on the shafts, if the keys are designed properly, and so on. For a multi-reduction gear drive system design, even though each gear set designed by the Gear AMD is perfect, the total reduction ratio of the multiple gear sets must meet a required ratio. Furthermore, any interference, when they are assembled with
Figure 4.18  Design Evaluating Task-Object in Multi-Level AMD

Figure 4.19  Example of Multi-Level AMD Hierarchy

(note) multiple instances are required for * marked AMDs
shafts, must be examined in the evaluation process. The role of the design evaluation task-objects in multi-level AMDs becomes more complicated and requires a decision-making ability, see Figure 4.18. Consequently, the design evaluation task-object is no longer a subclass of the TaskAssistant class that uses only formalized knowledge, but should be a subclass of the ControlPerformer class. In Figure 4.19, a possible AMD hierarchy of a parallel axes gear drive system design is given as an example. This implementation is under investigation at the present time.

4.6 Implementation of AMD Development Kit in Smalltalk

Although all necessary task-objects and AMDs are readily available, engineers may not be able to put them together to construct their design automation systems all by themselves. Therefore, the AMD Development Kit (AMDDK) has been implemented to help an engineer play with task-objects, build the ControlPerformer subclasses, and represent their design knowledge. The AMDDK is basically a user interface program which runs on top of the task-objects. It has features of registering new task-objects, configuring the AMD with desired task-objects, and monitoring how the AMD is working. Figure 4.20 shows the appearance of the AMDDK with the history-monitoring option turned "on". In the left window all communicating activities between the task-objects and the detailed performance are displayed while in the right window all design parameters and their changes during design procedures are displayed.
Figure 4.20 Basic Screen of AMD Development Kit

Figure 4.21 Class Registering Dialog Boxes
The first step to use the AMDDK is developing the subclasses of

*ControlPerformer* class, such as the *InitialDesigner* and *ReDesigner*. The actual
coding for these task-objects is done under the editor of a language. The AMDDK
provides the training facility for a selected Artificial Neural Network and rule
editing facility for developing the *ReDesigner* task-object. The next step is
registering the available task-objects to the list including the above task-objects,
such as the subclasses of the *TaskAssistant* class, *MaterialProperty*, *Geometry*,
*Rating*, and the subclasses of the *MethodPerformer* class. Registering is done by
selecting a class category in the menu "Register" and following through the dialog
boxes. A few dialog boxes are shown in Figure 4.21 as an example. Task-objects
can be removed from the list through the similar procedures, so that only necessary
ones are in the current list.

After registering the task-objects, an AMD is initialized by selecting an item
"New designer" from the menu "Designer". A user only needs to enter the name of
a new AMD and the AMDDK will initialize it automatically. With this step, an
empty AMD, namely only a skeleton of AMD, is created. Therefore, it must be
configured with the proper task-objects for each design task. They will be chosen
from the registered task-objects through the "Config" menu. The items of the
configuration are "User input list", "Initial design group", "Material group",
"Geometry", "Rating group", and "Re-design group". As examples, a user input
list entry screen is shown in Figure 4.22 and an initial design group configuration
dialog box is shown in Figure 4.23. An engineer builds the AMD through this
Figure 4.22 User Input Parameter Entry Dialog Box

Figure 4.23 Initial Design Group Dialog Box
configuration procedure with the selected task-objects. Once the AMD is configured, it is registered automatically to the list and can be used to build other AMDs. Any task-objects in the configuration may be changed as needed.

When the AMD is requested to perform a task, the AMDDK will pop-up a user input dialog box to receive the design inputs as shown in Figure 4.24. An engineer can decide the user input parameters when the AMD is configured (see Figure 4.22). The AMDDK displays those items in the dialog box. At this time, the AMD is already activated and this dialog box is requested by the class InputReceiver. The performance history monitoring and parameter monitoring windows are managed by the class Recorder which displays messages or values sent by task-objects. The partial displays of the performance history and the design parameters are shown in Figure 4.25. The entire lists displayed in those motoring windows can be found in Appendix B. Figure 4.26 illustrates how the AMDDK and other task-objects are related in their performances.

Although several parts of the AMDDK, such as the NetTrainer and Reporter objects are currently under implementation, the AMDDK successfully demonstrated its abilities as a development tool. When the whole AMDDK is completed as planned, engineers will spend the minimum efforts to develop their own design automation systems. They know how design is done since the AMD is configured with their choice of task-objects and furnished with their own design data and knowledge. The design data of the AMD will be grown as it performs a task and the knowledge of the AMD can be refined based on its performance. The
Figure 4.24 User Input Dialog Box

Figure 4.25 Design Outputs Displayed in Monitoring Windows
Figure 4.26 Relationship between AMDDK and AMD in Performance
AMD can be also used as a training tool for new design engineers. It provides
detailed task performance history from which the novice design engineers may
obtain expert knowledge within a reasonable time period without having actual
experience. They may try to change any embedded knowledge in a system to learn
the effect of the different knowledge.

Task-objects available currently are the only ones required to configure a
Gear AMD that will be discussed in the following chapter. All task-objects and
AMDs can be individually commercialized as long as they are developed within the
given guidelines. Various kinds of task-objects can be developed by different
software developers for the same task so that engineers will choose the one which
fits best to their application. If the performance of the selected one is not
satisfactory, it can be simply replaced by a better one to improve the performance.
Chapter 5

AMD Application to Gear Design

5.1 Organizing Design Procedures

The gear design task has been chosen as an example of a mechanical component design to demonstrate the applicability of an Artificial Mechanical Designer (AMD). Gears which transmit rotary motion from one shaft to another can be found in nearly every imaginable machine in the real world [55] and have been in use for over three thousand years [53]. Gears are ordinary machine elements that play a major role in providing the best performance in every machine. Designing gears requires making many complex decisions. The knowledge to make these decisions, an engineer's black box, has been established from an accumulation of past experiences and will be enhanced and adjusted from each new design experience as in all other mechanical design tasks. The first step in developing a Gear AMD using the AMD Development Kit (AMDDK) is organizing the design procedures so that they match the AMD construction shown in Figure 4.1. In Figure 5.1, one model of a Gear AMD construction obtained from an engineer who has more than 15 years design experience in industry is illustrated, which matches well the AMD construction as anticipated. This chapter will be focused on how to develop a Gear AMD and furnish it with an engineer's knowledge rather than discussing how a gear design is done.
Figure 5.1 Model of Gear AMD and Design Knowledge Sources
The first step in the figure is knowing the design requirements, namely a customer supplied input values. Usual input values include transmitted power, gear ratio, pinion input speed, service factors, and center distance limit. In the real world practice, only about 5 percent of the designs require a center distance limitation. Service factors of pitting resistance and bending strength may be replaced by application factors and other relevant values, such as life requirements and reliability factors, if preferred. In any case, the first three parameters are the essential ones and must be supplied by the customer.

The second step is to determine the initial design parameters based on the given design constraints. One method to obtain gear data consists of two consecutive steps which are to refer to a standard company catalog to identify an applicable center distance and accessing a company design data to estimate either the diametral pitch or the numbers of teeth. The rest of the initial design parameters can then be calculated with these estimated values. Looking up a product catalog can be viewed as finding the closest cluster in an LVQ algorithm while accessing the past design data can be compared with estimating output pattern in a GDR algorithm. Accordingly, the initial design can be emulated using the task-object, \textit{LVQwithMultiGDRNet} (reference to Section 4.3).

The next step is to evaluate a proposed design. In the figure, AGMA (American Gear Manufacturers Association) standards 6010-E88 [60] and 2001-B88 [59] are used for the material properties. AGMA 201.02 [62] and related formulas found in references [53] through [58] are used for geometry calculations. AGMA
2001-B88 and 908-B89 (for I and J factors) [61] are used for the power rating calculations. All of these steps depend upon formalized AGMA standards and formulas. The task-objects of these steps are assumed to be readily available.

The last step in the procedure is to change the design parameters when the evaluation has failed. This is accomplished by the engineer's decision making knowledge and can be emulated using the task-object, ForwardInference. Unlike catalog data or past design data, the redesign knowledge is not organized in a certain fashion. Rather, it is randomly memorized and acts intuitively upon a particular design situation. Therefore, careful attention is required when this knowledge is represented as the rule-like knowledge to be used in an inference engine. By assuming all necessary task-objects including the subclasses of the ControlPerformer are already available, only the design data preparation and the rule-base organization will be discussed in detail in the remainder of this chapter.

5.2 Preparing Design Data for Artificial Neural Network Task-Object

Applying an Artificial Neural Network requires two steps of preparation for an engineer, preparing a data pattern file and training a network. In a data pattern file, the existing design data are organized in the form of a pattern containing the input values and corresponding target output values. These data patterns serve as a set of initial knowledge to an Artificial Neural Network. A product catalog [69] obtained from a local gear manufacturing company is used as the data patterns for the first network, the LVQ algorithm. The catalog contains
Figure 5.2 Original Catalog Training Patterns

Figure 5.3 Transformed Training Patterns
three input parameters: transmitted power, gear ratio, and pinion input speed. In addition, the catalog also includes the model number which implies an applicable center distance for a given condition. These model numbers are used as pattern labels (or class labels) and will become the output of an LVQ network. The data of the selected models are plotted in Figure 5.2, where the patterns belonging to one model are scattered along the axes of gear ratio and pinion input speed. To apply the LVQ algorithm, the original three dimensional patterns are transformed and mapped onto a two dimensional pattern space as shown in Figure 5.3. Detailed implementation about the data transformation can be found in reference [28]. In Figure 5.4, examples of the data in a product catalog and the format of a data pattern file are shown.

The data sets for the GDR algorithm were also obtained from the same company, which are unavailable to public. These data are established from the company design history and organized to match the data in the catalog. Each data set is applicable to only a matching model. Consequently, the same numbers of data pattern files must exist for all models found in the catalog. The input values in these pattern files are the same as the ones for the LVQ algorithm and the diametral pitch is used as a target output value. The important consideration while preparing the data pattern files is how to arrange the available data in terms of their row order. The order of attributes in a row of data also affects the training results. Several trials may be necessary before finding the one that properly organizes the data into a usable form.
### Product Catalog (for MARK II model)

<table>
<thead>
<tr>
<th>Pattern File</th>
<th>Input Pattern</th>
<th>Torque</th>
<th>Model# (Output Pattern)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.18</td>
<td>352.0</td>
<td>400</td>
<td>MARK100S</td>
</tr>
<tr>
<td>1.50</td>
<td>302.0</td>
<td>400</td>
<td>MARK100S</td>
</tr>
<tr>
<td>1.69</td>
<td>275.0</td>
<td>400</td>
<td>MARK100S</td>
</tr>
<tr>
<td>1.79</td>
<td>261.0</td>
<td>400</td>
<td>MARK100S</td>
</tr>
<tr>
<td>2.04</td>
<td>232.0</td>
<td>400</td>
<td>MARK100S</td>
</tr>
<tr>
<td>2.18</td>
<td>218.0</td>
<td>400</td>
<td>MARK100S</td>
</tr>
<tr>
<td>2.50</td>
<td>189.0</td>
<td>400</td>
<td>MARK100S</td>
</tr>
<tr>
<td>2.88</td>
<td>161.0</td>
<td>400</td>
<td>MARK100S</td>
</tr>
<tr>
<td>3.00</td>
<td>155.0</td>
<td>400</td>
<td>MARK100S</td>
</tr>
<tr>
<td>3.44</td>
<td>131.0</td>
<td>400</td>
<td>MARK100S</td>
</tr>
<tr>
<td>3.76</td>
<td>118.0</td>
<td>400</td>
<td>MARK100S</td>
</tr>
<tr>
<td>4.00</td>
<td>109.0</td>
<td>400</td>
<td>MARK100S</td>
</tr>
<tr>
<td>4.55</td>
<td>91.3</td>
<td>400</td>
<td>MARK100S</td>
</tr>
<tr>
<td>5.00</td>
<td>80.6</td>
<td>400</td>
<td>MARK100S</td>
</tr>
<tr>
<td>5.66</td>
<td>67.0</td>
<td>400</td>
<td>MARK100S</td>
</tr>
<tr>
<td>6.36</td>
<td>56.5</td>
<td>400</td>
<td>MARK100S</td>
</tr>
</tbody>
</table>

**Figure 5.4 Product Catalog and Data Pattern File Samples**
The quality of training depends heavily on not only how data patterns are arranged but also on how the net parameters are selected. In the case of the GDR algorithm, the learning rate and momentum rate are set to 0.9 and 0.7 respectively as the default values in the AMDDK. These values are the typical ones for most applications [24]. However, it is necessary to adjust them to achieve a higher quality training result based on trial-and-error. According to reference [28], the learning rate can be selected between 0.95 and 0.99 and the momentum rate should not be higher than 0.95 to prevent being trapped in a local error minimum. These suggestions may not be adequate for all design data types. Therefore, much of the effort must be invested to find the values that provide the best training quality for a certain design data type. Figure 5.5 shows the format of a GDR net-file as a sample.

After networks are trained, the gear initial design task-object is developed by adding several specific methods as guided in Section 4.4. This task-object will be a subclass of the InitialDesigner and a possible name is GearInitialDesigner. The required method for a subclass of the InitialDesigner is the net-input-pattern method. This method returns a set of values which will be used as an input pattern for the LVQ network. These values are available from the parameter user-input-set and they must be arranged in the order used to prepare a data pattern file. Since the original data was transformed while training the LVQ network, the method transformed-pattern is also required. This method returns a set of values transformed from the original input pattern, i.e., the returns of the
Figure 5.5 Format of GDR Net Data Files

net-input-pattern method. The adjust-net-output method is also necessary to convert an estimated diametral pitch to a standard value. This method has all the standard diametral pitch values (or commonly used ones in industry [53][55]) in it and returns the nearest one to an estimated value, namely an output of the GDR network. An optional method default-initial-values is added to set normal pressure
netInputPattern
    | pattern |

    pattern := Array new: 3.
    pattern at: 1 put: (userSet at: 'Gear Ratio').
    pattern at: 2 put: (userSet at: 'Transmitted Power').
    pattern at: 3 put: (userSet at: 'pinion Input Speed').
    |pattern |

transformedPattern
    | mG hP np a c d tx1 tx2 pattern |

    mG := userSet at: 'Gear Ratio'.
    hP := userSet at: 'Transmitted Power'.
    np := userSet at: 'pinion Input Speed'.
    a := mG * hP * 100 / np.
    c := 10 * (mG exp) * (hP raisedToInteger: 3).
    d := (np squared) * (np sqrt).
    b := c / d.
    tx2 := 3 * a.
    tx1 := 5 * (a sqrt) * (b sqrt).

    pattern := Array new: 2.
    pattern at: 1 put: tx1.
    pattern at: 2 put: tx2.
    |pattern |

netOutputAdjust
    | c pn model |

    pn := netSet at: 'Normal Diametral Pitch'.
    model := netSet at: 'Model Number'.
    self task: 'Adjusting Normal Diametral Pitch based on standard'.
    (pn := (self adjustPn: pn rounded)) isNil
        ifTrue: [ 'self fatalError: 'Normal Diametral Pitch Exceeds Limit' ].
    self store: 'Normal Diametral Pitch' value: pn.
    self task: 'Finding proper Center Distance from Model Number'.
    (c := (self centerDistanceOn: model)) isNil
        ifTrue: [ 'self fatalError: 'Center Distance Not Found' ].
    self store: 'Center Distance' value: c.
    |true |

Figure 5.6 Sample Smalltalk Code for Selected Methods of GearInitialDesigner Class
angle, standard helix angle, and AGMA quality number whose initial values are normally determined by known default values. Other specific methods may be added to help the above methods complete their tasks. The sample Smalltalk code are given in Figure 5.6 for the selected specific methods.

5.3 Representing Design Knowledge for Inference Engine Task-Object

Rules are engineer's decision making knowledge represented in the form of symbolic and numeric expressions. The first step of developing rule methods for the GearReDesigner class, a subclass of the ReDesigner, is the constructing of a rule tree. A portion of the rule tree used in this application is introduced in Figure 5.7 as a sample. There are 20 condition-rules, 10 decision-rules, and 9 goal-rules in the entire rule-tree. Once a rule tree is completed, each rule in the tree will be converted to one of the three rule methods as guided in the Section 4.4. The Smalltalk code for the selected rules in the sample rule-tree, i.e. C01, C02, C04, D01, and G01, is listed in Figure 5.9. There are two requirements for rules. First, the first rule must be a condition rule and its conditional fact must be "evaluation failed". Since "evaluation failed" is the given fact from the DesignController, it is required that at least one rule (the first rule is recommended to reduce searching time) must have this fact as its conditional fact. Second, each goal rule must derive the fact "done" which is the goal fact in this implementation. If a goal rule does not derive this fact, an inference engine will continue to search another goal rule even after a decision is made. Reference to Figure 5.7 and 5.8, one possible
Figure 5.7 Sample Rule-Tree of Gear Redesign Knowledge
ruleC01
  `rule
  id: 'C01'
  fact: [ #(evaluation failed) isFact ]
  action: [ (rating ratingPoint) <= 1.0
    ifTrue: [ #(rating is under) ]
    ifFalse: [ #(rating is over) ]]
  explain: 'Rating difference is checked'
  why: 'Evaluation failed'

ruleC02
  `rule
  id: 'C02'
  fact: [ #(rating is under) isFact ]
  action: [ (geometry centerDistance) < (userSet at: 'Center Distance
    ifTrue: [ #(center distance is under limit) ]
    ifFalse: [ #(center distance is over limit) ]]
  explain: 'Center distance is checked'
  why: 'Rating is less than or equal to transmitted power'

ruleC04
  `rule
  id: 'C04'
  fact: [ #(center distance is over limit) isFact ]
  action: [ (pinionMaterial hardness) >= 555
    ifTrue: [ #(hardness reached limit) ]
    ifFalse: [ #(hardness is under limit) ]]
  explain: 'Hardness is checked'
  why: 'Center distance is greater than or equal to limit'

ruleD01
  `rule
  id: 'D01'
  fact: [ #(rating is over) ]
  action: [ #(decrease face width) ]
  explain: 'Decrease face width'
  why: 'Rating is more than 5% over transmitted power'

ruleG01
  `rule
  id: 'G01'
  fact: [ #(decrease face width) isFact ]
  action: [ geometry faceWidth:
    (((rating pittingResistanceRatingPoint reciprocal
      * (geometry faceWidth)) max:
      ((rating bendingStrengthRatingPoint squared) reciprocal
        * (geometry faceWidth))).
    #(done) ]

Figure 5.8 Sample Smalltalk Code for Rule Methods
chaining is as follows; "evaluation failed" - "rating is over" - "decrease face width" - "done".

Rules can access any evaluating task-objects, MaterialProperty, Geometry, and Rating instances, in order to obtain numerical values. Especially, the goal rules can access those instances to change the value of a selected design parameter. Once all rule methods are developed, the method add-all-rules needs to be added to the GearReDesigner class. This method simply adds each one of rules to an instance of an inference engine task-object based on their orders in the rule tree. A gear set consists of a pinion and a gear. Therefore, two instances of the MaterialProperty class are necessary for each one of them. Accordingly, the method set-material-assistant needs to be added, in which each instance of the MaterialProperty class is set to a local parameter, pinion-material or gear-material. Whenever, more than one material is used in the AMD, this method is mandatory for a subclass of the ReDesigner class.

5.4 Configuring Gear AMD with Selected Task-Objects

Before requesting a task of a Gear AMD, it must be configured with the available task-objects using the AMD Development Kit (AMDDK) introduced in Section 4.6. A configuration is done through several steps that appear in the menu "Config". The first step is setting up the user input parameter list. The names of those parameters should be entered as their full name, such as "Transmitted Power" and "Center Distance Limit". These names will be used in a user input dialog box
(see Figures 4.22 and 4.24). The next step is for the initial designer group which includes selections of an initial design task-object, a network algorithm task-object, and a net data file name (see Figure 4.23). For the MaterialProperty task-object, a component name, grade, and Brinell Hardness Number will be asked as initial values for each material selected. Then, the Geometry and Rating task-objects need to be selected and they are the GearParallerExternal and AGMA2001B88 classes. The class GearParallerExternal will request a task from the AGMA201.02 in order to find all tooth proportional geometry. The class AGMA2001B88 will request a task from the AGMA908B89, a subclass of the Geometry, for l and J factors of a gear and a pinion. The last step is the selecting of the redesign task-object, the GearReDesigner in this case. In Figure 5.9, a fully configured Gear AMD is illustrated, where the gray colored objects and data are the ones developed by an engineer. Once all task-objects are configured, a Gear AMD is ready to perform a design. The sample outputs of the Gear AMD are given in Appendix B for selected gear design tasks.

The configurations of a Gear AMD can be changed by selecting a different task-object for each category. The knowledge source of a task-object, no matter if it is informal or formal, can be replaced by difference knowledge. Design data sets used in one company may be different from others. Redesign knowledge may also vary by engineers or companies. Furthermore, the gear power rating procedures may vary, since there exists a couple of different methods. Consequently, the design outputs produced by a Gear AMD will be different and based on the
(Note) 1: training  2: consulting

Figure 5.9 Fully Configured Gear AMD
methods selected and the knowledge used. Any single or all data patterns used to train an Artificial Neural Network may be changed if the design outputs are not satisfactory or if better data or new data are available. Also the redesign rule methods can be modified, deleted, added, and reconstructed as needed.

The AMD is designed to be configured in any shape and to be furnished with any kind of design knowledge. The AMD is not a magic box which would produce a perfect design from nothing. It is like a small child, a future mechanical engineer, who does not know how to design and does not have any knowledge at the moment. However, a small child has the ability to learn, so does the AMD.
Chapter 6
Conclusions

6.1 Summary

Mechanical design tasks usually require both formalized knowledge and non-formalized knowledge. Rules of thumb, common sense, and intuition are all examples of non-formalized knowledge. Such knowledge is associated with an engineer's decision making ability and derived from many years of design experience. Developing a mechanical design automation system which resembles the performance of a human engineer is deeply related to how non-formalized knowledge can be represented and maintained. In addition, what kind of design methodology is used by a human engineer in design practice is also an important issue in this research area.

In this project, the object-oriented programming paradigm, along with Artificial Intelligence techniques, are employed to develop the framework of a mechanical design automation system. Mechanical design procedures, perhaps the most general procedures, are decomposed into small task units to take advantages of the object-oriented programming. As a result, a task-performing object (or task-object) and an Artificial Mechanical Designer (AMD) are conceptualized and successfully implemented. Task-objects are designed to share common behavior and communication protocol. Accordingly, they can request that tasks be performed by others and can receive task-orders from others.
Furthermore, multiple Artificial Intelligence techniques are applied in the form of task-objects in this project while most of the existing design systems utilize only a single technique. Again it must be emphasized that a mechanical design process consists of many small tasks which identify their own characteristics. The only way of emulating mechanical design tasks as performed by a human engineer is by incorporating both AI and non-AI techniques. The results of this application were satisfactory but more techniques need to be experimented with in order to achieve the ultimate goal.

An AMD is not a prefixed design system. It is an abstracted framework which can be configured with a choice of task-objects, namely any kind of design strategies and methodologies that can be applicable. Consequently, the design system built on an AMD have more flexibility. They are compatible in terms of their behavior so that they can be replaced easily by better or newer task-objects without changing the system itself. Expendability is only limited by an engineer's design assignments and it can be done by adding new task-objects featured with more specific and overwritten methods. The MethodPerformer task-objects can be furnished with various types of knowledge which will evolve as it performs more designs.

When this AMD implementation becomes widely accepted and further enhanced, engineers will gain powerful development tools which are easily adaptable, expandable, refinable, and exchangeable, yet have intelligence to learn. Another advantage includes the engineer's ability to investigate their valuable time
to develop more effective design methodologies, as well as optimizing the available
solutions. As a result, novice designer may obtain accumulated expert knowledge
from a well built AMD without having the time consuming real experience.
Eventually, the company will achieve not only design consistency, but more reliable
design output.

6.2 Limitations of Current Implementation and Future Plans

Although the concept of a task-object and an Artificial Mechanical Designer
(AMD) could enhance deficiencies in the current methodologies used by other
design automation systems, there exist several limitations which must be overcome
in future works. The most critical limitation lies in the object-oriented
programming (OOP) languages themselves. The first choice of an OOP language
for the AMD implementations was the Common Lisp Object System (CLOS).
CLOS is the object-oriented extensions to the list processing language LISP which
has been the most widely utilized programming language in the field of Artificial
Intelligence research. Although, the CLOS specifications was standardized in June
1988 [68], the only available CLOS implementation at the time of beginning this
work was PCL (Portable Common Loops) developed at the Xerox Palo Alto
Research Center (Xerox PARC). The PCL implementation includes only partial
functions of the programmer interface specified in Chapter 2 of the CLOS
specification. After several months of investigation, it was found that the PCL
could not support full implementations of the AMD concept.
The next programming language experimented with was C++ released in July 1983 by AT&T Bell Laboratories [14]. C++ is also an object-oriented extension to the procedural language C. Because the C language is most widely used in the system-level programmer's community, C++ is intentionally designed to be fully compatible with the C language. As a result, the class definition in C++ became more like the structure definition in the C language, including some added object-oriented features, such as inheritance and information hiding. Furthermore, classes become user defined data types, just like "integer" or "string". When arguments are passed to an object, the data types of arguments must match the ones defined in the method interface. If any of the passed arguments do not match, a compiling error will occur. This is also true for the argument passed between the regular functions. Consequently, a task-object is able to communicate only with pre-selected task-objects, which is obviously not a desirable case.

The third language tried was Smalltalk/V®, an implementation derived from Smalltalk-80™ by Digitalk, Inc. [9]. Smalltalk is a pure object-oriented language and includes all the beneficial capabilities of the object-oriented approach. The primary deficiency is that it does not support multiple inheritance, in other words, an object can have only a single direct superclass. The lack of multiple inheritance mechanism caused some programming overhead in many task-objects, especially the subclasses of the TaskAssistant class in this implementation. Had this been available, a class hierarchy of task-objects could have been organized more
efficiently and logically. Nevertheless, Smalltalk was chosen and the AMD was successfully implemented.

The abstract class *TaskObject* needs to be enhanced by means of a communication capability. With the current implementation, a task-requestor object can not send a task-order if it does not know a passing argument list prior to requesting a task. The mechanism to query a task-performer object of what arguments need to be passed and inspect whether all required arguments are available in a parameter list is not included. This feature is desired in order to develop a more robust task-objects. A task-object would not need to know a passing argument list *priori* and it could be able to ask for such a list before sending an actual task-order if this feature was implemented.

As already pointed out in Chapter 4, subclasses of the *ControlPerformer* class should be reimplemented in order that they can accept any *MethodPerformer* task-object. The current implementation limits only a pre-selected methodology to each subclass. It was intentionally done to prevent any confusions by means of task names (or procedure descriptions) in constructing the AMD with multi *ControlPerformer* task-objects. However, in the future implementation it is advisable to correct this limitation, so that an AMD will obtain even further flexibility in its construction. Other future plans should be addressed developing more efficient inference engine task-objects and enhanced design evaluation task-objects. One example is that current *Rating* class is not adequate for a
multi-level AMD. This class needs to be enhanced so that it can accept both formalized and non-formalized knowledge.

Some of features of the AMD Development Kit, such as the classes NetTrainer and RuleEditor, are under investigation and not included in this implementation. These features are not a part of an AMD but would be convenient tools if available. Nevertheless, the concept of a task-object and an AMD, along with the AMD Development Kit, demonstrated an applicability to mechanical design tasks and will hopefully dominate over methodologies used in traditional design automation systems.
REFERENCES


24. EEAP484: *Adaptive Pattern Recognition and Neural Networks*, Class Lecture Note, Case Western Reserve University (1990)


57. Lynwander, Peter, Gear Drive Systems: Design and Application, Marcel Dekker, Inc. (1983)


64. AGMA Standard 120.01, "Gear-Cutting Tools: Fine- and Coarse-Pitch Hobs," American Gear Manufacturers Association (1975)


Appendix A

Class Definitions of Artificial Mechanical Designer
A.1 Class Hierarchy of AMD Task-Objects

Recorder
TaskObject
TaskPerformer
ControlPerformer
    DesignController
    InputReceiver
    InitDesigner
    ReDesigner
MethodPerformer
    SingleNeuralNet
    LVQNet
    GDRNet
    LVQwithMultiGDRNet
    ForwardInference
TaskAssistant
MaterialProperty
    Steel
Geometry
    AGMA20102
    AGMA908B89
    GearParallelExternal
Rating
    AGMA2001B88

NetObject
    NetPoint
    NetSpace
    NetCluster
    NetNode
    NetLayer
    Network

ExpertObject
    ExpertRule
    ExpertFact

(Note) The base classes of the AMD are printed in bold type-faces.
A.2 Definitions of AMD Base Classes

Recorder

A Recorder is used to display text to the designated TextPane. All instances share the same TextPane which is defined by the AMDDK when it requests a design to the DesignController. Therefore, it is not necessary to initialize the Recorder unless the TextPane needs to be changed.

Inherits From: Object

Inherited By: (None)

Named Instance Variables: (None)

Class Variables:

Pane
Contains the instance of the TextPane

Pool Dictionaries: CharacterConstants

Class Methods:

initialize: aTextPane
One time initialization. Initialize the class variable Pane to aTextPane.

name: aName value: aValue
Record aName/aValue pair with ':' character to the Pane.
 aValue can be Number, String, Class, Boolean, OrderedCollection, Array.

record: aString1 sider: aString2
Record aString1 with aString2 to the Pane. aString2 can be nil.

recordAfterBlank: aString1 sider: aString2
Record aString1 with aString2 after one blank line to the Pane.
aString2 can be nil if it is not required.

recordBeforeBlank: aString1 sider: aString2
Record aString1 with aString2 to the Pane, then feed one blank line.
aString2 can be nil if it is not required.
Instance Methods:

name: aName array: anArray
  Record aName/anArray pair with ':' character to the Pane.

name: aName value: aValue
  Record aName/aValue pair with ':' character to the Pane.

record: aString1 sider: aString2
  Record aString1 with aString2 to the Pane.

recordAfterBlank: aString1 sider: aString2
  Record aString1 with aString2 after one blank line to the Pane.

recordArray: anArray
  Record a single array to the Pane. Recursible for multi level array display.
  Accept only a Number and String array.

recordBeforeBlank: aString1 sider: aString2
  Record aString1 with aString2 to the Pane, then feed one blank line.

TaskObject

Class TaskObject is the superclass of all task-objects. It is an abstract class defining the common protocol of task performance for all of its subclasses. This class has all default methods of the required methods which are required to its subclasses. Some of the default methods will cause a fatal error when a subclass does not include a required method which is mandatory.

Inherits From: Object

Inherited By: TaskPerformer ControlPerformer DesignController InputReceiver InitDesigner ReDesigner MethodPerformer ForwardInference LVQwithMultiGDRNet SingleNeuralNet LVQNet GDRNet

Named Instance Variables:

parameters
  Contains a Dictionary which holds return parameter name and value pairs
**recorder**
Contains an instance of the class Recorder

**fatalError**
Contains a fatal error message

**Class Variables:** (None)

**Pool Dictionaries:** (None)

**Class Methods:**

**initialize: aDictionary**
Initialize the receiver task-object with aDictionary which contains arguments.

**Instance Methods:**

**activateTaskObject: aTaskObject with: arguments**
Initialize aTaskObject, call the method perform, and answer its instance.

**fatalError: aString**
Open the fatal error message box containing aString as an error message and answer false. This method also records aString message to the record pane and causes the termination of the current design.

**initialize**
Initialize the arguments of the receiver task-object, recorder and parameters.

**inspectInput**
Answer true by default. Assuming the performer does not require argument inspection.

**perform**
Control the performance of the receiver task-object. Answer false if any fatal error occurred. Else answer true. This method sends the messages inspectInput and performTask.

**performTask**
Answer the fatal error by default. Assuming the performer does not have a performTask method which is a required method.

**record: aName value: aValue**
Record aName/aValue pair to the record pane.
requestTaskTitle
Record the request task title to the record pane.

requestTaskTo: aTaskObject with: anArguments
Initializes aTaskObject and send the message perform. Answer nil if aTaskObject returns false. Else, answer the returns of aTaskObject.

result: aString
Record aString with a prefix '>' as the result of a task to the record pane.

returns
Answer the return parameter dictionary of the receiver task-object, parameters.

searchValueFor: aName among: aDictionary
Answer the value whose key is aName in the return parameter dictionary. If not found, answer nil. This method is recursible when aName itself is a dictionary.

store: aName value: aValue
Store aName (parameter name) and aValue to the parameter dictionary.

task: aString
Record the task begin message aString with '...' to the record pane.

value: aName
Answer the value whose key is aName in the parameter dictionary. This method calls the method 'searchValueFor: aName among: aDictionary'.

TaskPerformer

This class is added to organize all ControlPerformer and MethodPerformer task-objects. No variables and methods are defined in this class.

Inherits From: TaskObject Object

Inherited By: ControlPerformer DesignController InitDesigner InputReceiver ReDesigner
**ControlPerformer**

Class *ControlPerformer* is the superclass of all the control-performer classes. It is an abstract class grouping the control-performer classes to separate from the method-performer classes which behave differently.

*Inherits From:* TaskPerformer TaskObject Object

*Inherited By:* DesignController InputReceiver InitDesigner ReDesigner

*Named Instance Variables:*

- fatalError (From class TaskObject)
- parameters (From class TaskObject)
- recorder (From class TaskObject)

`configs`  
Contains a Dictionary which holds a designer configuration set.

*Class Variables:* (None)

*Pool Dictionaries:* (None)

*Class Methods:* (None)

*Instance Methods:*

- **initialize:** aDictionary  
  Initialize the arguments of the receiver performer with aDictionary of input arguments. Answer nil with the fatal error if failed. Else, answer self.

- **inspectInput**  
  Inspect the passed input arguments and answer true if all arguments are found. Else, answer the fatal error.

**DesignController**

A *DesignController* is the main control class of all AMD classes. The instance of a *DesignController* will be created by the class *DesignerDeveloper* in AMD Development Kit or by the another instance of a *DesignController* which is
connected to it. The instances of this class is not intended to be created by any user methods. The methods of this class are all private and should not be redefined.

**Inherits From:**  
ControlPerformer TaskPerformer TaskObject Object

**Inherited By:**  
(None)

**Named Instance Variables:**

- fatalError
  - (From class TaskObject)
- parameters
  - (From class TaskObject)
- recorder
  - (From class TaskObject)
- configs
  - (From class ControlPerformer)

**userSet**
Contains a Dictionary which holds user input parameter and value pairs.

**initialSet**
Contains a Dictionary which holds initial design parameter and value pairs.

**historyPane**
Contains an instance of history recording **AMDTextPane** class.

**parameterPane**
Contains an instance of parameter recording **AMDTextPane** class.

**statusPane**
Contains an instance of status displaying **AMDGraphNode** class.

**Class Variables:**  
(None)

**Pool Dictionaries:**  
(None)

**Class Methods:**  
(None)

**Instance Methods:**

**initialize:** aDictionary
Initialize the arguments of the receiver task-object with aDictionary which holds arguments. Answer nil with the fatal error if failed. Else, answer the receiver.

**inspectInput**
Inspect the passed input arguments and answer true if all arguments are found and valid. Else, answer the fatal error.
myName
Answer the full name of the DesignController.

performTask
Control design procedures based on designer configurations. Requests tasks from the InputReceiver, an initial designer class, and a redesigner class and activates material classes, a geometry class, and a rating class. Answer false if any fatal error occurred. Else, answer true.

receiveReturnsFrom: anInstance (or aCollectionOfInstances)
Receives the returns from anInstance (a performer) and store them to the receiver's return parameter dictionary. This method also records name and value pairs of the returns to the parameter record pane.

recordParameters: aKey inOrder: aCollectionInOrder
Record the values of aKey in the receiver's return parameter dictionary to the parameter record pane. The order of recording is based on aCollectionInOrder containing return parameters in their order, only if aCollectionInOrder is not nil.

requestInitDesign
Request a task from an initial designer class with arguments containing user input and initial design configurations. Answer the returns from its instance.

requestReDesign
Request a task from a redesigner class with arguments containing user input, initial design results, material instances, geometry instance, rating instance, and redesign configurations. Answer the returns from its instance.

requestUserInput
Request user inputs from the InputReceiver with arguments containing user input list and a designer name. Answer the returns from its instance.

InputReceiver

A InputReceiver is used to receive user inputs from a user. An instance of this class is created by the DesignController class only. If several DesignController is connected in multi-level AMD, the instance is created only once by the top level DesignController class.

Inherits From: ControlPerformer TaskPerformer TaskObject Object
Inherited By:  (None)

Named Instance Variables:

fatalError  (From class TaskObject)
parameters  (From class TaskObject)
recorder    (From class TaskObject)
configs     (From class ControlPerformer)

itemList
  Contains a collection of a user input list.

Class Variables:  (None)

Pool Dictionaries:  (None)

Class Methods:  (None)

Instance Methods:

initialize: aDictionary
  Initialize the arguments of the receiver task-object with aDictionary which holds arguments. Answer nil with the fatal error if failed. Else, answer the receiver.

inspectInput
  Inspect the passed input arguments and answer true if all arguments are found and valid. Else, answer the fatal error.

myName
  Answer the full name of the InputReceiver.

performTask
  Open a user input dialog box and receive user inputs. Answer true if all user inputs are received. Else, answer the fatal error.

InitDesigner

Class InitDesigner is the superclass of all the initial designer classes. It is an abstract class defining the common protocol for all of its subclasses. This class has all default methods of the required methods to its subclasses. Some of the default
methods will cause a fatal error if a subclass does not implement any required method which is mandatory.

Inherits From: ControlPerformer TaskPerformer TaskObject Object

Inherited By: (None)

Named Instance Variables:

fatalError (From class TaskObject)
parameters (From class TaskObject)
recorder (From class TaskObject)
configs (From class ControlPerformer)

userSet
Contains a dictionary which holds user input values.

netSet
Contains a dictionary which holds neural net output values.

Class Variables: (None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:

initialize: aDictionary
Initialize the arguments of the receiver task-object with aDictionary which holds arguments. Answer nil with the fatal error if failed. Else, answer the receiver.

inspectInput
Inspect the passed input arguments and answer true if all arguments are found and valid. Else, answer the fatal error.

myName
Answer the InitDesigner's full name.

netInputPattern
Answer the empty collection by default which will cause the fatal error.
**netOutputAdjust**
Answer true by default. Assuming neural net outputs will not be adjusted.
Store all values in the netSet to the parameter dictionary.

**performTask**
Request a task from a neural net class indicated in the designer configurations
and generate initial design values. Answer false if any fatal error occurred.
Else, answer true.

**transformedPattern**
Answer nil by default. Assuming net input pattern transformation is not
required.

---

**ReDesigner**

Class *ReDesigner* is the superclass of all the re-designer classes. It is an abstract
class defining the common protocol for all of its subclasses. This class has all
default methods of the required methods to its subclasses. Some of the default
methods will cause a fatal error if a subclass does not implement any required
method which is mandatory.

**Inherits From:**  
*ControlPerformer TaskPerformer TaskObject Object*

**Inherited By:**  
(None)

**Named Instance Variables:**

- **fatalError**  
  (From class *TaskObject*)
- **parameters**  
  (From class *TaskObject*)
- **recorder**  
  (From class *TaskObject*)
- **configs**  
  (From class *ControlPerformer*)

- **userSet**
  Contains a dictionary which holds user input values.
- **initialSet**
  Contains a dictionary which holds initial design values.
- **materials**
  Contains a collection which contains instances of material classes.
- **geometry**
  Contains an instance of a geometry class.
rating
Contains an instance of a design rating class.

rule
Contains an instance of the class ExpertRule.

ruleList
Contains a collection of rules.

Class Variables:

Decisions
Contains a collection of decisions made throughout the entire design.

Pool Dictionaries: (None)

Class Methods:

decisions
Answer the collection of decisions made, Decisions.

initializeDecisions
One time initialization. Initialize the class variable Decisions.

Instance Methods:

addAllRules
Add the rule 'ruleC01' by default to the ruleList. This method causes a fatal error since no rule methods are implemented by its subclass.

addRule: aRule
Add aRule to the ruleList.

initialize: aDictionary
Initialize the arguments of the receiver task-object with aDictionary which holds arguments. Answer nil with the fatal error if failed. Else, answer the receiver.

initializeMaterial
Reinitialize the variable materials with the first MaterialProperty instance in the variable materials by default. Assuming only one material is used in the entire design procedure.

inspectInput
Inspect the passed input arguments and answer true if all arguments are found
and valid. Else, answer the fatal error.

**myName**
Answer the full name of a redesigner class.

**performTask**
Initialize all available rules including the final rule ruleG00 and request redesign decision to a selected expert system. Answer false if any fatal error occurred. Else, answer true. This method records all actions taken for a redesign decision.

**ruleC01**
Answer the first condition rule by default. This method causes a fatal error since the subclass does not have the initial condition rule to search. The fact of this rule is #(evaluation failed) which is the known fact. This rule will derive the new fact #(design failed) by default.

**ruleG00**
Answer the final rule - the last rule to be searched. This method is common for all redesign subclasses. This method causes the design failure since the fact of this rule is #(design failed).

**MethodPerformer**
This class is added to organize all MethodPerformer task-objects such as artificial neural networks and rule-based expert systems. No variables and methods are defined in this class.

*Inherits From:* TaskPerformer TaskObject Object

*Inherited By:* (None)

**TaskAssistant**
This class is added to organize all task-objects which use formalized design knowledge. It has only one instance variable changeFlag. The changeFlag is initially defined as 'false' and will be changed to 'true' when a redesigner class accesses any parameter to change the current value.

*Inherits From:* TaskObject Object
Inherited By:       Geometry Material Property Rating

Named Instance Variables:

fatalError         (From class TaskObject)
parameters         (From class TaskObject)
recorder           (From class TaskObject)

changeFlag
    Contains a boolean expression.

Class Variables:   (None)

Pool Dictionaries: (None)

Class Methods:     (None)

Instance Methods:

changeFlag
    Answer the changeFlag.

initialize
    Initialize the instance variable the changeFlag to false and access its superclass's initialize method.

resetChangeFlag
    Reinitialize the changeFlag to false.

Geometry

This class is added to organize all task-objects whose role is geometry calculations. The method ‘performTask’ is defined in this class and always answer true by default. All subclasses are not supposed to have this method unless required.

Inherits From:     TaskObject Object

Inherited By:      (None)
Named Instance Variables:

fatalError   (From class TaskObject)
parameters   (From class TaskObject)
recorder     (From class TaskObject)
changeFlag   (From class TaskAssistant)

Class Variables:   (None)

Pool Dictionaries: (None)

Class Methods:    (None)

Instance Methods:

groupName
   Answer the groupTitle of the Geometry class which is used when the return parameters are displayed in the record pane.

performTask
   Answer always true by default. This method is defined as a dummy method to prevent task termination.

MaterialProperty

Class MaterialProperty is the superclass of all material task-objects such as Steel and Bronze. It is an abstract class defining the common protocols of task performance for all of its subclasses. The method 'performTask' is defined in this class and always answers true by default. All subclasses are not supposed to have this method unless required.

Inherits From: TaskObject Object

Inherited By: (None)

Named Instance Variables:

fatalError   (From class TaskObject)
parameters   (From class TaskObject)
recorder     (From class TaskObject)
changeFlag   (From class TaskAssistant)
component
    Contains a String for a component name.

grade
    Contains an Integer for a material grade.

hardness
    Contains a Number for Brinell Hardness Number.

modulusOfElasticity
    Contains a Number for a modulus of elasticity.

poissonsRatio
    Contains a Number for a Poisson's ratio.

heatTreatment
    Contains a String for heat treatment recommendation.

Class Variables:    (None)

Pool Dictionaries:  (None)

Class Methods:

property
    Initialize the receiver material and set all known properties.

Instance Methods:

component
    Answer the component.

grade
    Answer the grade.

grade: anInteger
    Set the grade to anInteger and set the changeFlag to true.

groupTitle
    Answer the groupTitle of the MaterialProperty class which is used when the
    return parameters are displayed in the record pane.

hardness
    Answer the hardness.
hardness: aHardness
    Set the hardness to aHardness and set the changeFlag to true.

heatTreatment
    Answer the heatTreatment.

initialize: aDictionary
    Initialize the receiver material based on the component name, grade, and
    hardness.

modulusOfElasticity
    Answer the modulusOfElasticity.

performTask
    Answer always true by default. This method is defined as a dummy method to
    prevent task termination.

poissonsRatio
    Answer the poissonsRatio.

returns
    Answer all return parameters in the form of a Dictionary whose keys are
    parameter names.

returnsInOrder
    Answer a collection which holds all return parameters in their proper order.

Rating

    This class is added to organize and represent the task-objects which use
    formalized design knowledge.

Inherits From: TaskObject Object

Inherited By: (None)

Named Instance Variables:

fatalError (From class TaskObject)
parameters (From class TaskObject)
**recorder** (From class TaskObject)

**changeFlag** (From class TaskAssistant)

**ratingLog**
Contains a boolean expression.

**ratingPoint**
Contains a Number for the ratio between applicable value and required value.

**Class Variables:** (None)

**Pool Dictionaries:** (None)

**Class Methods:** (None)

**Instance Methods:**

**groupTitle**
Answer the groupTitle of the Rating class which is used when the return parameters are displayed in the record pane.

**ratingLog**
Answer the ratingLog, true for successful rating or false for failed rating.

**ratingPoint**
Answer the ratingPoint.
Appendix B

Sample Outputs of Gear AMD
B.1 Performance History Outputs

DESIGN STARTED by Gear AMD

REQUEST TASK
<<Requestor>> AMD Development Kit
Initializing task-object ...

<<Performer>> Gear AMD Design Controller
Inspecting arguments ...
> Can not find user input

REQUEST TASK
<<Requestor>> Gear AMD Design Controller
Initializing task-object ...

<<Performer>> Gear AMD User Input Receiver
Inspecting arguments ...
Performing task ...
Requesting input from user ...
> Received following user input
  Required Gear Ratio: 3.75
  Transmitted Power: 70
  pinion Speed: 1500
  Cs1: 1.2
  Kaf: 1.2
  Center Distance Limit: 8
> Performed successfully
TASK DONE by Gear AMD User Input Receiver
Performing task ...

REQUEST TASK
<<Requestor>> Gear AMD Design Controller
Initializing task-object ...

<<Performer>> Gear AMD Initial Designer
Inspecting arguments ...
Performing task ...
Preparing arguments for LVQWithMultiGDRNet ...
Preparing net input pattern ...
  pattern: (27.059774 52.5 )

REQUEST TASK
<<Requestor>> Gear AMD Initial Designer
Initializing task-object ...

<<Performer>> LVQ with multi GDR Neural Net
Inspecting arguments ...
Performing task ...
Preparing arguments for LVQ Neural Net ...

REQUEST TASK
<<Requestor>> LVQ with multi GDR Neural Net
Initializing task-object ...

<<Performer>> LVQ Neural Net
Inspecting arguments ...
Performing task ...
Initializing LVQ Neural Net ...
numOfCenters: 4
dimOfCenter: 2
Finding the closest cluster to the pattern ...
> Pattern ID (netOutput) found is;
pattern id: MARK2507
> Performed successfully
TASK DONE By LVQ Neural Net

Preparing arguments for GDR Neural Net ...

REQUEST TASK
<<Requestor>>  LVQ with multi GDR Neural Net
Initializing task-object ...

<<Performer>> GDR Neural Net
Inspecting arguments ...
Performing task ...
Initializing GDR Networks ...
  number of hidden layers: 1
  number of each layer nodes: (1 1 1 )
  input normalizing factors: (26.293827 1458.2357 9172.7861 )
  output normalizing factors: (53.103672 )
  weight 111: -1.302616
  weight 112: -2.499349
  weight 113: 3.420832
  threshold 11: -0.160146
  weight 121: -0.789884
  weight 122: -1.951506
  weight 123: 2.860698
  threshold 12: -0.519491
  weight 131: -9.326554
  weight 132: 1.538308
  weight 133: -3.884463
  threshold 13: 2.554089
  weight 211: -2.396706
  weight 212: -2.275168
  weight 213: -3.444286
  threshold 21: 2.521279
  hidden nodes: (3 )
Normalizing net inputs ...
  normalized pattern: (1.4261902e-1 4.8003213e-2 1.6352719e-1 )
Denormalizing net outputs ...
  net outputs: (6.3448478 )
> Performed successfully
TASK DONE By GDR Neural Net

> Performed successfully
TASK DONE By LVQ with multi GDR Neural Net

Adjusting Normal Diametral Pitch based on standard ...
  Normal Diametral Pitch: 6.0
Finding proper Center Distance from model number ...
  Center Distance: 7.0
Setting default parameter values ...
> Performed successfully
TASK DONE By Gear AMD Initial Designer

ACTIVATE Task-Object
<<Requestor>>  Gear AMD Design Controller
Initializing task-object ...

<<Performer>> Steel Property
Inspecting arguments ...
Performing task ...
> Activated successfully
> Performed successfully
TASK DONE By Steel Property

ACTIVATE Task-Object
<<Requestor>> Gear AMD Design Controller
Initializing task-object ...

<<Performer>> Steel Property
Inspecting arguments ...
Performing task ...
> Activated successfully
> Performed successfully
TASK DONE By Steel Property

ACTIVATE Task-Object
<<Requestor>> Gear AMD Design Controller
Initializing task-object ...

<<Performer>> Parallel-External Gear Geometry
Inspecting arguments ...
Performing task ...
Calculating basic geometry ...
> Performed successfully
TASK DONE By Parallel-External Gear Geometry

ACTIVATE Task-Object
<<Requestor>> Parallel-External Gear Geometry
Initializing task-object ...

<<Performer>> AGMA 201.02: tooth proportional geometry
Inspecting arguments ...
Performing task ...
Activating task-object ...
> Performed successfully
TASK DONE By AGMA 201.02: tooth proportional geometry

ACTIVATE Task-Object
<<Requestor>> Gear AMD Design Controller
Initializing task-object ...

ACTIVATE Task-Object
<<Requestor>> AGMA 2001888: Gear Power Rating
Initializing task-object ...

<<Performer>> AGMA 908889: I and J factors
Inspecting arguments ...
Performing task ...
Activating task-object ...
> Performed successfully
TASK DONE By AGMA 908889: I and J factors
AGMA 2001B88: Gear Power Rating
Inspecting arguments ...
Performing task ...
Calculating AGMA power rating ...
> Rating point found is;
  ratingPoint: 9.0440368e-1
> Rating failed
> Performed successfully
TASK DONE By AGMA 2001B88: Gear Power Rating

REQUEST TASK
<<Requestor>> Gear AMD Design Controller
Initializing task-object ...

<<Performer>> Gear AMD Re-Designer
Inspecting arguments ...
Performing task ...

ACTIVATE Task-Object
<<Requestor>> Gear AMD Re-Designer
Initializing task-object ...

<<Performer>> Forward-Channing Inference Engine
Inspecting arguments ...
Performing task ...
Chaining facts and Searching for the goal rule ...
> Goal is found
> Performed successfully
TASK DONE By Forward-Channing Inference Engine

Fact: Known fact is received
  Action: Redesign is started
Fact: Evaluation failed
  Action: Rating difference is checked
Fact: Rating is less than or equal to transmitted power
  Action: Center distance is checked
Fact: Center distance is less than limit
  Action: Rating difference is checked again
Fact: Rating is less than 15% under transmitted power
  Action: Rating difference is checked again
Fact: Rating is less than or equal to 10% under transmitted power
  Action: Helix angle is checked
Fact: Helix angle is less than 15 degrees
  Action: Pac rating is checked
Fact: Pac rating is failed
  Action: Helix angle is checked for gear type
Fact: Helix angle is 0 degrees
  Action: Aspect Ratio is checked
Fact: Aspect ratio is under limit
  Action: Increase face width
Fact: Goal is found
  Action: ReDesign is done
> Performed successfully
TASK DONE By Gear AMD Re-Designer

Calculating AGMA power rating ...
> Rating point found is;
ratingPoint: 1.2749896
> Rating failed

REQUEST TASK
<<Requestor>> Gear AMD Design Controller
Initializing task-object ...

<<Performer>> Gear AMD Re-Designer
Inspecting arguments ...
Performing task ...

ACTIVATE Task-Object
<<Requestor>> Gear AMD Re-Designer
Initializing task-object ...

<<Performer>> Forward-Chaining Inference Engine
Inspecting arguments ...
Performing task ...
Chaining facts and Searching for the goal rule ...
> Goal is found
> Performed successfully
TASK DONE By Forward-Chaining Inference Engine

Fact: Known fact is received
  Action: Redesign is started
Fact: Evaluation failed
  Action: Rating difference is checked
Fact: Rating is more than 5% over transmitted power
  Action: Decrease face width
Fact: Goal is found
  Action: ReDesign is done
> Performed successfully
TASK DONE By Gear AMD Re-Designer

Calculating AGMA power rating ...
> Rating point found is;
  ratingPoint: 1.0330618
> Rating succeeded

ACTIVATE Task-Object
<<Requestor>> Parallel-External Gear Geometry
Initializing task-object ...

<<Performer>> AGMA 201.02: tooth proportional geometry
Inspecting arguments ...
Performing task ...
Activating task-object ...
> Performed successfully
TASK DONE By AGMA 201.02: tooth proportional geometry

> Performed successfully
TASK DONE By Gear AMD Design Controller

DESIGN DONE By Gear AMD
B.1 Design Parameter Outputs

Gear AMD

>> USERSET
   Required Gear Ratio: 3.75
   Center Distance Limit: 8
   Ksf: 1.2
   pinion Speed: 1500
   Transmitted Power: 70
   Csf: 1.2

>> INITIALSET
   Normal Diametral Pitch: 6.0
   Helix Angle: 0
   Center Distance: 7.0
   Face Width: 3.0
   Normal Pressure Angle: 20
   AGMA Quality Number: 8

>> MATERIAL PROPERTY 1
   Material: Steel
   Component: pinion
   Grade: 1
   Hardness: 150
   Modulus Of Elasticity: 30000000
   Poissons Ratio: 0.3
   Allowable Contact Stress: 140800
   Allowable Bending Stress: 39300
   Heat Treatment: Through Hardened

>> MATERIAL PROPERTY 2
   Material: Steel
   Component: gear
   Grade: 1
   Hardness: 300
   Modulus Of Elasticity: 30000000
   Poissons Ratio: 0.3
   Allowable Contact Stress: 120000
   Allowable Bending Stress: 36000
   Heat Treatment: Through Hardened

>> GEOMETRY
   pinion Teeth Number: 18
   gear Teeth Number: 68
   Gear Ratio: 3.7778
   Normal Diametral Pitch: 6
   Normal Pressure Angle: 20.0
   Helix Angle: 0.0
   Center Distance: 7.1667
   Operating Center Distance: 7.1667
   Face Width: 3.0
   Net Face Width: 1.0
   Aspect Ratio: 1.0
   pinion Pitch Diameter: 3.0
   gear Pitch Diameter: 11.3333
   pinion Operating Pitch Diameter: 3.0
   gear Operating Pitch Diameter: 11.3333
   pinion Outside Diameter: 3.3333
   gear Outside Diameter: 11.6667
pinion Base Diameter: 2.8191
gear Base Diameter: 10.6498
Pressure Angle: 20.0
Operating Pressure Angle: 20.0
Operating Normal Pressure Angle: 20.0
Base Helix Angle: 0.0
Operating Helix Angle: 0.0
Diametral Pitch: 6.0
Circular Pitch: 0.5236
Base Pitch: 0.492
Normal Base Pitch: 0.492
Axial Pitch: 0
Addendum: 0.1667
Dedendum: 0.2083
Clearance: 0.0417
Working Depth: 0.3333
Whole Depth: 0.375
Fillet Radius: 0.05
Circular Tooth Thickness: 0.2618
Width of Top Land: 0.0417
Normal Circular Pitch: 0.5236

>> STANDARD RATING
Rating Point: 0.9044
Pitting Resistance Rating: 63.3083
Bending Strength Rating: 108.1734
Application Power: 63.3083
Elastic Coefficient: 2290.6039
Dynamic Factor: 0.9679
Load Distribution Factor: 1.1721
Hardness Ratio Factor: 1.0
Temperature Factor: 1.0
Surface Condition Factor: 1.0
Size Factor: 1.0
Rim Thickness Factor: 1.0
AGMA Quality Number: 8
I Factor: 0.1043
pinion J Factor: 0.24
gear J Factor: 0.2816
Load Sharing Ratio: 1.0
Transverse Contact Ratio: 1.6666
Axial Contact Ratio: 0.0

>> DECISIONSET
decisions:
 ("Increase face width")
reDesignLog: true

>> STANDARD RATING
Rating Point: 1.275
Pitting Resistance Rating: 89.2493
Bending Strength Rating: 149.6787
Application Power: 89.2493
Elastic Coefficient: 2290.6039
Dynamic Factor: 0.9679
Load Distribution Factor: 1.2471
Hardness Ratio Factor: 1.0
Temperature Factor: 1.0
Surface Condition Factor: 1.0
Size Factor: 1.0
Rim Thickness Factor: 1.0
AGMA Quality Number: 8
I Factor: 0.1043
pinion J Factor: 0.24
gear J Factor: 0.2816
Load Sharing Ratio: 1.0
Transverse Contact Ratio: 1.6666
Axial Contact Ratio: 0.0

>> DECISIONSET
decisions:
("Increase face width"
"Decrease face width")
reDesignLog: true

>> MATERIAL PROPERTY 1
Material: Steel
Component: pinion
Grade: 1
Hardness: 350
Modulus Of Elasticity: 30000000
Poissons Ratio: 0.3
Allowable Contact Stress: 140800
Allowable Bending Stress: 39300
Heat Treatment: Through Hardened

>> MATERIAL PROPERTY 2
Material: Steel
Component: gear
Grade: 1
Hardness: 300
Modulus Of Elasticity: 30000000
Poissons Ratio: 0.3
Allowable Contact Stress: 120000
Allowable Bending Stress: 36000
Heat Treatment: Through Hardened

>> GEOMETRY
pinion Teeth Number: 18
gear Teeth Number: 68
Gear Ratio: 3.7778
Normal Diametral Pitch: 6
Normal Pressure Angle: 20.0
Helix Angle: 0.0
Center Distance: 7.1667
Operating Center Distance: 7.1667
Face Width: 3.5
Net Face Width: 3.5
Aspect Ratio: 1.1667
pinion Pitch Diameter: 3.0
gear Pitch Diameter: 11.3333
pinion Operating Pitch Diameter: 3.0
gear Operating Pitch Diameter: 11.3333
pinion Outside Diameter: 3.3333
gear Outside Diameter: 11.6667
pinion Base Diameter: 2.8191
gear Base Diameter: 10.6498
Pressure Angle: 20.0
Operating Pressure Angle: 20.0
Operating Normal Pressure Angle: 20.0
Base Helix Angle: 0.0
Operating Helix Angle: 0.0
Diametral Pitch: 6.0
Circular Pitch: 0.5236
Base Pitch: 0.492
Normal Base Pitch: 0.492
Axial Pitch: 0
Addendum: 0.1667
Dedendum: 0.2083
Clearance: 0.0417
Working Depth: 0.3133
Whole Depth: 0.375
Fillet Radius: 0.05
Circular Tooth Thickness: 0.2618
Width of Top Land: 0.0417
Normal Circular Pitch: 0.5236

>> STANDARD RATING
Rating Point: 1.0331
Pitting Resistance Rating: 72.3143
Bending Strength Rating: 121.2773
Application Power: 72.3143
Elastic Coefficient: 2290.6039
Dynamic Factor: 0.9679
Load Distribution Factor: 1.1971
Hardness Ratio Factor: 1.0
Temperature Factor: 1.0
Surface Condition Factor: 1.0
Size Factor: 1.0
Rim Thickness Factor: 1.0
AGMA Quality Number: 8
I Factor: 0.1043
pinion J Factor: 0.24
gear J Factor: 0.2816
Load Sharing Ratio: 1.0
Transverse Contact Ratio: 1.6666
Axial Contact Ratio: 0.0