Parameterizing and capturing diagnostics information during design.

Roozbeh Rahimpour

University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation


https://scholar.uwindsor.ca/etd/2992
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
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700  800/521-0600
Parameterizing and Capturing Diagnostics Information During Design

by
Roozbeh Rahimpour

A Thesis Submitted to the Faculty of Graduate Studies and Research through the Department of Industrial and Manufacturing Systems Engineering in partial fulfillment of the requirements for the Degree of Master of Applied Science at the University of Windsor

Windsor, Ontario, Canada
May 1997
© 1997, Roozbeh Rahimpour
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-30975-4
Abstract

This thesis presents a new methodology for constructing a device's computer model to be used as the knowledge-base component of the model-based diagnostics expert system of the device. It is shown that parameterized functional and behavioural knowledge of a device could be applied to the diagnostics action to pin point the root cause of a fault. Four underlying concepts which are examined and deployed in this thesis to support the proposed methodology are: Parametric and feature based design, object oriented methodology for analysis and design, manufacturing as incorporating useful information into material, and using functional knowledge in diagnosing faults. The intended area of research explored in this project is Applications of Artificial Intelligence in manufacturing. To demonstrate the steps involved in implementing the proposed methodology, a prototype system is presented for diagnosing faults of a hair dryer based on its functional and behavioural parameters.
Dedications

To my wife for her patience, support and sacrifices,
To my family,
and
To others who consistently, confidently, and continuously strive
towards achieving excellence in what they do, and encourage others along the way.
Acknowledgments

I would like to thank my faculty advisor Dr. S. Taboun for his patience and words of advice and most importantly for his availability during odd hours of the day. I would also like to thank my supervisor at EDS of Canada Mr. Dennis Goodman for his words of encouragement along the way and making the required resources available.
# Table of Contents

ABSTRACT iii

DEDICATIONS iv

ACKNOWLEDGMENTS v

LIST OF TABLES viii

LIST OF FIGURES ix

LIST OF ABBREVIATIONS x

1. INTRODUCTION 1

1.1 Challenges Experienced in Developing Expert Systems 2

1.2 Opportunity for Improvement 3

1.3 Statement of Objectives 4

2. LITERATURE REVIEW 6

2.1 The Diagnostic Process 6

2.2 Approaches to Expert Systems for Diagnostics 9

2.2.1 Evolution of Diagnostics Expert Systems 9

2.2.2 The First Generation: Rule-Based 10

2.2.3 The Second Generation: Model-Based 14

2.2.4 Third Generation: Case-Based Diagnostics Expert Systems 18

2.2.5 Hybrid Diagnostics Expert Systems 22

2.3 Adopting Strategies for Diagnostics and Expert Systems for This Thesis 27

2.4 Search for Similar Research 28

3. METHODOLOGY 30

3.1 Contributing Notions 30

3.1.1 Manufacturing as Embedding Information into Material 30

3.1.2 Feature Based or Parametric Design 31

3.1.3 Model-Based Diagnostics 31

3.1.4 Object Orientation 32

3.2 Parameterized-Objects Diagnostics Expert System (PODES) 32

3.2.1 Independent and Intelligent Component Classes 34

3.2.2 Relationships 36

3.2.3 Faults 39
<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Merits and Demerits of Rule-based, Model-based, Case-based Systems</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Knowledge Sources and Their Contents</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>Component Class Analysis</td>
<td>49</td>
</tr>
<tr>
<td>4</td>
<td>Relationship Class Analysis</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>Fault Class Analysis</td>
<td>52</td>
</tr>
<tr>
<td>6</td>
<td>State-Event Fault Analysis</td>
<td>53</td>
</tr>
<tr>
<td>7</td>
<td>Component Fault Analysis</td>
<td>53</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Life Cycle of the Computer Model of a Product or Device</td>
<td>4</td>
</tr>
<tr>
<td>Figure 2</td>
<td>An Example of Components and Dataflow in Rule-based System</td>
<td>11</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Main Components of a Model-based System</td>
<td>16</td>
</tr>
<tr>
<td>Figure 4</td>
<td>An Example of Dataflow and Main Components of a Case-Based System</td>
<td>21</td>
</tr>
<tr>
<td>Figure 5</td>
<td>A Generic Independent Intelligent Component Object</td>
<td>35</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Four Examples of Relationships Between Objects</td>
<td>37</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Network of Independent and Intelligent Objects</td>
<td>38</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Suggested Information Flow Pre and Post-Fault</td>
<td>39</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Reactional Dependencies Between Components</td>
<td>40</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Proposed Fault-Tree for Device Fault Diagnostics</td>
<td>42</td>
</tr>
<tr>
<td>Figure 11</td>
<td>High Level System Architecture for PODES</td>
<td>44</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Physical Representation for Hair Dryer</td>
<td>48</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Logical Representation for Hair Dryer</td>
<td>48</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Component Object Model for Hair Dryer</td>
<td>51</td>
</tr>
<tr>
<td>Figure 15</td>
<td>State Diagram for Hair Dryer</td>
<td>52</td>
</tr>
</tbody>
</table>
List of Abbreviations

CAD: Computer Aided Design / Computer Aided Drafting
CAM: Computer Aided Manufacturing
CID: Computer Integrated Diagnostics
CIM: Computer Integrated Manufacturing
FBD: Feature Based Design also referred to as Parametric Design
OO: Object Oriented
OOM: Object Oriented Methodology
PODES: Parameterized-Objects Diagnostics Expert System
1. INTRODUCTION

In an attempt to improve productivity and efficiency of the diagnostic process, expert systems for fault diagnostics have been the focus of considerable research in the past few decades [1, 2, 3]. In fact, intelligent fault diagnostic systems have been recognized as "the most common application of knowledge-based (expert) systems in the manufacturing environment" [4, 5].

Expert systems are also used in many other industries for performing a variety of tasks such as: design for manufacturing [6], intelligent manufacturing systems [7, 8], training of maintenance personnel [9, 10], intelligent CAD tools [11, 12, 13], medical diagnosis [14, 15], maintenance in CIM environment [16], machine and equipment fault diagnosis [1, 2, 15, 17, 18, 19, 20, 21], diagnosis of power station control systems and power distribution networks [22, 23], process maintenance and monitoring [24, 25, 26], probing hydraulic faults [27], tender evaluation [25], and machine condition monitoring [28]. Majstorovic in [1] presents a detailed analysis of more than 60 expert systems; Pfau-Wagenbauer and Nejdl in [22] present a tabular review of eleven real-time expert systems designed and developed for alarm processing in power distribution networks.

The intended domain of research in this thesis is expert systems for device fault diagnostics. The focus of this thesis in the domain of device fault diagnostics is the concurrent development of functional and geometrical knowledge-bases for a device. The functional knowledge-base is used in the diagnostics process whereas the geometrical knowledge-base is used throughout manufacturing processes of the device. The reason for focusing on the two development activities concurrently is to overcome some of the
challenges currently experienced in developing model-based expert systems for fault diagnostics.

1.1 Challenges Experienced in Developing Expert Systems

Traditionally, expert systems for device fault diagnostics have been developed after the device enters its utilization or productive stage in its life cycle. Gathering the relevant information for creating the knowledge base of the diagnostics expert system would exhibit a great challenge. As the device ages, the amount of experiential knowledge about the device grows as more faults are dealt with, and therefore, the challenge of collecting and formulating facts about the device would require more effort. Furthermore, due to lack of sufficient experiential knowledge during the early productive stage of the device, diagnostics would rely more on functional and behavioural knowledge about the device.

The greatest effort in developing expert systems is spent on collecting, formulating, and representing the domain knowledge. Expert shells have been developed as an attempt to significantly reduce the development activities involved in expert systems. Expert shells are expert systems but with an empty knowledge base. Expert shells provide the user with pre-defined knowledge representation and formulation statements. Hence, developing an expert system by using an expert shell is reduced to stating and representing facts about the domain. However, the challenge of collecting and structuring the facts about the domain still remain.

Another contributing factor to the challenge of developing diagnostics expert systems is the centralized approach to inferencing. The domain knowledge and logical procedures are compiled into central structures called the knowledge base and the inference engine in traditional diagnostics expert systems. Modifying or adding new
facts about the domain or logical procedures in the central structures would require a significant effort.

The challenges of information gathering and overcoming the centralized inferencing approach in developing diagnostics expert systems present the opportunity for developing an alternative methodology.

1.2 Opportunity for Improvement

Figure 1 shows the four main stages in the life cycle of the computer model of a product or device from the intelligent manufacturing point of view. The CAD model of a product is released for manufacturing after completion of the design stage. The model would include geometrical dimensions and tolerances (GD&T). The manufacturing systems generate process planning and machine codes for the numerically controlled machines, such as milling machines and lathes. After the part is manufactured, it is generally inspected to ensure quality of the manufacturing processes as well as the part. In the third stage, the inspection data is compared against the designed dimensions to determine if the part is within the intended tolerances. In other words, the main activity in stage 3 is to ensure that the inputs to the first stage match the outputs from the second stage, within specified tolerances. Integrated computer aided products exist which would facilitate sharing of data amongst the three stages through the CAD model without requiring data conversion nor translation.

However, the fourth stage in the product life cycle, namely product fault diagnostics remains to be a disintegrated activity in the product life cycle, as no evidence was found in the literature indicating otherwise. Throughout the literature review no reference was found to a product nor a methodology which would allow capturing and storing diagnostics worthy information available during design, manufacturing, and inspection for the purpose of diagnosing potential faults in the product. The dotted lines
connecting diagnostics to the intelligent manufacturing cycle in Figure 1 indicate the lack of integration of device fault diagnostics with the other three stages.

![Diagram showing the life cycle of the computer model of a product or device]

**Figure 1: Life Cycle of the Computer Model of a Product or Device**

Integrating the development of the functional knowledge-base with designing the product would allow capturing functional and behavioural information to be used in diagnosing potential faults of the product when using a model based diagnostics expert system. The Computer Integrated Diagnostics (CID) approach would facilitate sharing of diagnostics information in between different stages of the life cycle of the product’s computer model.

### 1.3 Statement of Objectives

Creating the CID environment is not the objective of this thesis. However, there are a number of underlying concepts which need to be established before a CID strategy could be implemented. Two of the underlying concepts are investigated and presented in this thesis. The first objective of this thesis is to develop a methodology for capturing and structuring a device’s functional knowledge in its functional knowledge-base. The proposed methodology will address two concerns: first to develop virtual models of physical components of a device, and second to incorporate parameterized diagnostics
knowledge in the virtual models. Diagnostics knowledge would include data as well as procedures. Embedding diagnostics procedures in the virtual model of a component would enable the component to conduct limited diagnosing on its own operations, inputs, and outputs. Hence, the inferencing capabilities and operations are distributed throughout the computer model of the device.

The second objective of this thesis is to develop a tool based on the proposed methodology. Reflecting the underlying concepts presented in the methodology, the tool is called Parameterized-Objects Diagnostics Expert System (PODES). The designer would use this tool to create the functional knowledge-base of a device. PODES could be used simultaneously with a CAD environment to develop the geometrical model of the device. The designer would be able to change the parameterized attribute values of the components in the device’s functional knowledge-base. The diagnostician would use the functional knowledge-base to diagnose faults by identifying a complaint about the device.

In brief, with the intention of significantly reducing the efforts and resources that it takes to design and develop a diagnostics expert system for a device, this thesis focuses on Parameterizing and capturing a device’s functional information to create a functional model of the device which could be used to diagnose potential faults in that device. A prototype of PODES is developed to demonstrate its operations and effectiveness.
2. Literature Review

The literature review for this thesis was conducted to satisfy two objectives: first, to adopt the effective diagnostics strategy as well as the effective expert system approach in support of the thesis objectives, second to identify similar research work in the domain of diagnostics expert systems. To achieve the two objectives, the literature review focused on two main activities: first to investigate the diagnostics process and its information requirements, second to establish the state of the art in the domain of diagnostics expert systems in order to identify challenges and opportunities for further research. The literature review is presented in this chapter under three sections. The first section reviews the diagnostics process and how it is defined. The second section presents different types of expert systems referenced in the literature. The final section presents the literature on research that contributed and bears resemblance to this thesis.

2.1 The Diagnostic Process

Webster’s New World Dictionary defines *diagnosis* as: “the act or process of deciding the nature of a diseased condition by examination of the symptoms; a careful examination and analysis of the facts in an attempt to understand or explain something.” Other definitions for diagnosis are: learning to bring the right pieces of knowledge together at the right time” [14]. “the process of determining the fault or faults responsible for a set of symptoms” [17], “formulation and investigation of hypotheses about the malfunctioning device” [18], comparison of current operating parameters (states) against known (ideal) values and determining the cause(s) of less than optimal performance [2, 29]. How the diagnosis process is defined and interpreted determines the analysis
approach towards diagnostic expert systems as well as the type of knowledge that one needs to have available to complete a diagnosis. For example, if the diagnosis is defined as "knowing the difference" [4] between normal and abnormal behaviours of a device, then one needs to have more functional knowledge about the internal structure of the device and how its constituent parts interact. On the other hand, if the diagnosis is defined as responding to symptoms and trying to determine the cause(s) of the symptoms, then the diagnosis decision making relies more on rules of thumb and less on fundamental (functional and behavioural) knowledge about the device.

At first, diagnostic action involves a continuous and integrated process of decision making: recalling previous scenarios and trying to find the closest match for the existing situation: categorization and identification of the possible state of affairs in the environment. A suggested three step approach to diagnosis would involve: the diagnostician first analyzes the situation, then selects a goal or develops one or a number of hypothesis and follows it by planning for testing and proving or disproving each of the hypothesis [4]. This diagnostic strategy is experience-based [5] and relies on a seasoned diagnostician to have been "conditioned" [4] over time for the task. If the situation is the first time occurrence of an incident, by solely relying on this strategy the diagnostician would fail to identify the fault and the cause of it.

An alternative strategy would be causal reasoning based on representation of the behaviour of the physical world [4]. This representation is comprised of mathematical relationships between types (classes) of components of the physical world. What is taken for granted in the physical world and is essential in causal reasoning is the intuitive knowledge about the normal and abnormal state of affairs. The intuitive knowledge implicitly takes advantage of the decomposition of the physical world. Therefore, the level of the intuitive knowledge determines how far the physical world needs to be decomposed into its sub-components.
To explain how a fault took place, one needs to trace the situation back until a satisfactory cause is identified. The fact that a component broke down in a device is not necessarily the satisfactory cause of the fault. But rather, what lead to the break-down of the component is the explanation that is of interest. Whether the component (or the operator) was operating within the as-designed and normal setting is the question requiring an explanation. Knowledge of previous incidents could be even misleading in these situations. For example, many causes could lead into a headache. Relying on the patience history primarily as the basis for diagnosis could have severe consequences if the true cause is not the usual one. [4]

Bublin and Kashyap in [18] present a more detailed approach to diagnostic action than the three step approach by Rasmussen in [4]. They present a five step strategy for diagnostic problem solving which they summarize as “formulation and investigation of hypothesis about the malfunctioning device”: (i) formulation of a problem by analyzing the situation, making observations, and developing plausible hypothesis; (ii) developing expectations for each of the hypothesis; (iii) selecting the hypothesis with highest expectations for leading to the cause; (iv) collecting data to evaluate if the hypothesis is supported or not; (v) evaluating the hypothesis and reformulating to repeat step (i) in case of indecisiveness of the hypothesis. The five steps are repeated as many times as required until the cause is identified.

Based on the above review, the diagnostic process is both experienced-based and knowledge intensive [5]. To successfully respond to a faulty condition, one requires background information about the domain as well as familiarity with the internal structure of the system or device.
2.2 Approaches to Expert Systems for Diagnostics

Regardless of the goal (whether to provide an explanation for abnormal behaviour of the device, to compensate the system as a response to emergency, or to correct the disorder in the system), diagnostic reasoning involves establishing some logical association between "a device's abnormal behaviour and the failure" [18]. But, the effective strategy for constructing the logical association depends on the nature of the knowledge required for the diagnosis.

Ayeb, Marquis and Rusinowitch in [30] distinguish between two main approaches for developing expert diagnostic systems. They call one approach "empirical-association-based" and the other "model-based." Other researchers have made a similar differentiation towards categorizing expert systems for diagnostics [4, 5, 15, 17, 20, 21, 22]. Yet, a more recent approach to expert systems called case-based reasoning has drawn much attention and has proved to be effective and efficient in processing knowledge for diagnostics decision making. Case-based reasoning is also an associative approach because it utilizes empirical knowledge in the diagnostics process. However, cases in case-based are much more comprehensive in capturing the total state of the fault than rules in a rule-based system.

2.2.1 Evolution of Diagnostics Expert Systems

Based on the type of knowledge compiled in the knowledge-base, expert systems for diagnostics have been classified into three main categories of rule-based, model-based, and case-based. Watson in [31] refers to the third generation as case-based, and Storr and Wiedmann in [20] refer to the third generation as the hybrid of the rule-based and model-based approaches. Storr and Wiedmann were not incorrect by titling the third generation hybrid-systems. Their paper was published five years prior to Watson's. However, to categorize approaches to diagnostics expert systems, both Watson and Storr
and Wiedmann used the term _generation_ instead of _type_, perhaps intending to imply the concept of evolution and advancement in expert systems for diagnostics.

2.2.2 The First Generation: Rule-Based

Although work on DENDRAL, the world’s first expert system, started in the 1960s, the first generation of diagnostics expert systems bloomed in mid 1970s with products such as Internist, MYCIN, and Prospector. DENDRAL was used for describing chemical compounds. Internist and MYCIN were developed to diagnose internal diseases and bacterial infections. Prospector was developed in 1978 to assist geologists to locate energy and mineral deposits.[32]

Some of the rule-based expert systems for diagnostics identified by Majstorovic in [1] are: SAGE-SD, COLLAC, ATREX, AL/X, EIDEMON, PDS, REACTOR, DELTA, FALCON, NDS. ARBY (for diagnosing electronic failures) in [17] and MindMeld (for diagnosing hydraulic faults) in [27] are additional examples of rule-based systems for diagnostics; DELTA, TURBOMAC, and DIPLOMA are presented in [5] as examples of rule-based expert systems for process design and life-cycle optimization. Figure 2 displays a simplistic system layout including the data flow and main components for a rule-based system.

Knowledge Requirements in Rule-Based Systems

The type of knowledge used in developing rule-based systems is of the experiential nature [5, 20], also referred to as empirical, shallow, surface, compiled, and low-road knowledge [21]. It consists mainly of the rules of thumb and intuitions that an experienced diagnostician has learned over the years in his or her field of expertise [5, 17]. This knowledge is very focused and specific to the field; therefore, providing extensive explanations about a fault in the domain is beyond the capability of rule-based
systems [5, 21]. The extents of the logical inferences which could be made by the inference engine is limited to the logical derivations of the factual information about the domain in the knowledge-base. As a result, rule-based systems are capable of effectively handling faults which have been hard coded in the knowledge-base, but only those. The effectiveness of the rule-based systems "often degrades quickly when faced with a problem just outside its realm of expertise" [21].

![Diagram](image-url)

Figure 2: An example of components and dataflow in rule-based system

Five different sources of expertise knowledge are identified by Majstorovic in [1]: (i) the domain expert, (ii) previously developed databases for maintenance planning, (iii) information about sensors or condition monitoring devices, (iv) technical documentation
including maintenance manuals, diagnostic handbooks, and operators manuals, and (v) maintenance reports containing expected fault conditions and their respective causes.

Knowledge Acquisition in Rule-Based Systems

A number of steps need to take place in preparation for incorporating the human knowledge into the knowledge-base of an expert system. The first step is identifying the individuals to fill the two crucial roles of the domain expert and the knowledge engineer [1, 5, 27]. The more important of the two roles is that of the domain expert. The success in developing the rule-based system and the effectiveness of the system in identifying faults and providing explanations are mainly attributed to the availability of the domain expert and the robustness of his or her experiences [5, 17, 27]. The domain expert should exhibit and willingly share his or her knowledge (with the knowledge engineer) including rules of thumb, short-cuts in diagnostics and quick-hit responses to the symptoms [1, 5].

The second role is that of the knowledge engineer who is charged with the responsibility of capturing and formalizing the expertise knowledge in preparation for constructing the knowledge base and the inference engine. The knowledge engineer could represent a potential bottleneck in the development cycle. Thus, the knowledge engineer’s role is also critical: he or she should be able to ask the right questions and make effective observations in order to capture the expert’s knowledge. The next step after capturing the knowledge is formalizing it by effective knowledge representation schemes [1].

Knowledge Representation and Inferring in Rule-Based Systems

Depending on the size and complexity of the rule-based system, there are three system architectures for the inference engine. In the first scheme, the inference engine is physically a part of the knowledge base and the knowledge base includes both the declarative facts and logical instructions. This scheme is used in smaller and simpler
systems. Due to the small size of the knowledge-base, little degradation in system response time is observed. As the systems' knowledge-base grows, the concept of separating the knowledge-base from the rule-base (inference engine) becomes more compelling. The second architecture scheme separates the knowledge base and the rule-base into two separate physical structures. As the size of the knowledge-base and the complexity of the procedures grow, division of knowledge and logic reveals greater efficiency. The third system architecture scheme divides the knowledge-base even further into a number of smaller knowledge-bases. By structuring types of knowledge into separate structures the most suitable representation and search strategies could be applied to each type of knowledge in each structure. Thus, the third architecture scheme achieves a greater overall system efficiency perhaps at the cost of greater system complexity.[1]

Discussion on the First Generation Systems

As the first general attempt to computerize the decision making process, the first generation expert systems were developed to emulate the human intelligence as much as possible within a narrow field in a specific domain; within their respective well defined narrow scope, many of the rule-based systems proved to be effective decision making tools [5]. However, the inherent characteristics of the rule-based expert systems create three bottlenecks. First, the narrow field of the expertise limits their wide applicability within the same domain. Second, the expertise knowledge is hard coded in rule-based systems. Hence, expanding the knowledge base becomes a major and perhaps costly maintenance activity. On the other hand, "an incomplete rule-base system provides little value because a missing rule could halt the reasoning process" [33]. Third, collecting and formulating expertise knowledge for creating the knowledge base is perhaps the most challenging tasks in developing rule-based diagnostics expert systems.
2.2.3 The Second Generation: Model-Based

Based on what was found in the literature in [15] and [18], it could be stated that the practical venturing with model-based systems for diagnostics goes back at least to the late 1970s. Recent papers on expert systems for diagnostics indicate a strong focus on this approach amongst researchers in this field [5, 15, 18, 19, 23, 24]. Bublin and Kashyap present a comprehensive theoretical coverage of the deep knowledge approach to expert systems for diagnostics [18]. A few applications and examples of model-based systems are presented in [5, 15, 19, 23, 24].

The main component of a model-based system is the computer model of the physical device or system undergoing diagnosis. Functional, behavioral, and design knowledge about the device are used to form the computer model and to create a logical connection between the virtual device’s components. By definition, in model-based systems a fault takes place when a discrepancy is observed between the expected and observed system’s responses and behaviour [15, 17]. As a result, for model based systems diagnosis is defined as identifying and explaining the discrepancy between the intended and observed outcomes.

Generally, model-based systems perform their purpose by employing the following five steps in the way most suitable to the problem, although not necessarily in the sequence presented here. The first step in diagnosing the fault would be to localize it, although this might not be necessary depending on the structure and setup of the sensory mechanism. The second step is to develop hypothesis and select suitable tests and test cases to run through the virtual component or device (the computer model) in order to imitate the physical device and its states as closely as possible. The third step is to simulate the faulty input to the component which generated the fault. Dynamic simulation techniques are employed to generate the faulty state(s) of the device in the computer model. The test signal is propagated throughout the virtual model in an attempt to generate a total device response. The forth step is to compare the responses with the
intended response and observe discrepancies. In the final step, the explanation module of the model-based expert system creates an explanation on the cause of the fault, and suggests options for treatment.[17]

Figure 3 displays the main components of a typical model-based system.

**Knowledge Requirements in Model-Based Systems**

As the second generation of expert systems [20], model based systems rely on deeper knowledge in the domain. Deep knowledge is also known as fundamental domain knowledge, high-road causal, functional, or physical knowledge [21].

In model-based systems, the physical domain is modeled by using the functional and behavioral knowledge of the device. Four types of knowledge about the device are required for developing a computer model and establishing logical interconnections between its components. The four types of knowledge are: device’s design intent, inputs and expected outputs, operational states, and performance constraints. Information of this nature could be captured from the design documentation. In fact the designer could build the knowledge-base for a model-based system during the design phase of a device, provided that he or she uses a modern computer aided design software package. [15]

In addition to the functional knowledge about the device, there are three other types of knowledge required for diagnosing a device. Test procedures and cases are required for propagating test input values throughout the system during the simulation stage. Since not every fault could be tested and hence identified based on structural and behavioral knowledge of the device, certain assumptions need to be made about the device to further the diagnostic process. Applying the correct assumptions to the fault situation on hand requires certain knowledge. Knowledge of developing hypothesis and procedures for proving or negating the hypothesis is the third type of knowledge required for a model-based system.[15]
Figure 3: Main components of a model-based system

By employing techniques such as parametric design [11, 34, 35], feature-based component representation and design [12, 36, 37], and object-oriented design [7, 11, 38], modern computer aided design tools facilitate embedding functional and behavioural information (such as the design intent information, material properties and mechanics and boundary conditions as well as inputs and outputs to each component [38]) into the device’s computer model. The smart product models developed based on these techniques contain “the why and how of the design in addition to the what of the design” [38].

The Functional Model

Each constituent component or sub-system of a device or system could be identified by three attributes: its design intent and function, a set of operational constraints, and how it is related to other parts or components in the device. This is the basis of the computer model for the device to be used for simulating the device. In addition to imitating the device in the computer world, the computer model should depict what events take place in the device, which components of the device are involved in
each event, and timing of the involvement of different components in each event. Apart from what takes place in each component, the question remains whether the component produces or processes a product. The functional model of the device could be represented by four primary types of concepts: structural, products, behaviour, and functions.[18]

The structural concept entails physical grouping of components under a device and logical grouping of related components for the formation of a system under the device. Each system performs a specific task in the device. Devices could also be identified by the products that they produce or process. However, a product does not have to have a physical form or shape: a change of state in the device could be considered as a product as well. The intention behind assigning products to devices or their components is to be able to track the processes which they handle and the different states that they are transformed through. Each component is also identified by its transfer function: how it transforms its inputs to generate outputs. The set of transformations that a device performs on its inputs is called the device’s behaviour instead of its function. The role (used instead of component’s function to prevent confusion) of each component in the device is another identifying concept.[18]

Discussion on The Second Generation Systems

Although effective functional expert systems have been developed based on the pure model-based approach. Fink and Lusth in [21] and Punch in [14] state that the causal reasoning can not be a total replacement for rule-based systems. Model-based system are computationally demanding and expensive. To develop a comprehensive functional model of a device would require significant effort and resources; the more complex the device, the more expensive developing the functional model. Furthermore, the effectiveness of the model-based system depends on the level of detail embedded in the functional model [39].

Some of the inefficiencies of model-based systems, such as poor response time, could be significantly improved by integrating the model-based system with a shallow
knowledge system [22]. The model-based approach should be used to fill the gaps left by the heuristic or rule-based approach that uses shallower knowledge [14]; it should be used in a supporting role as opposed to the primary diagnostic technique [21]. But one should not lose sight of the fact that model-based systems “for well understood domains can be very effective and are a relatively mature and well understood technology” [40].

2.2.4 Third Generation: Case-Based Diagnostics Expert Systems

As an attempt to overcome the limitations of the first and second generation diagnostics expert systems, since the early 1980s considerable attention has been focused on the third generation approach: case-based reasoning (CBR) systems. In pursuit of a closer model for the human cognitive process, CBR is considered as the third generation of expert systems.[40]

Research work on how to take advantage of CBR in expert systems started in the mid 1980s; Stanfill and Waltz reported in 1986 on how they used CBR for “determining the correct pronunciation of a word from a store of words accompanied by their correct pronunciations” [41]. CLAVIER is the first reported commercial expert system which has been in use since 1990. It was developed by Lockheed to satisfy four primary concerns: to take advantage of successful cases, to reduce the pressure on experts, to leverage the expertise knowledge as corporate asset, and to help in training new personnel. In addition to satisfying these requirements, CLAVIER is also serving as a corporate data repository, transferring knowledge between the users.[40]

A case is comprised of facts as well as associated circumstances which was used at one point to identify and resolve a situation and could be used in identifying similar situations in the future [40]. To deal with a new situation, CBR systems attempt to find one or many similar cases dealt with in the past [42]. Solutions to already experienced problems are adapted to the current problem on hand. “Because CBR systems associate
features of a problem with a previously-derived solution to that problem, they are
classified as association-reasoning systems” [43], belonging to the same class of expert
systems as rule-based systems.

The underlying operating principle of CBR is analogy [31]. Webster’s New
World Dictionary defines analogy as “an explaining of something by comparing it point
by point with something similar, the inference that certain admitted resemblance imply
probable further similarity.” CBR systems match successfully resolved situations in the
past with the current situation on hand and apply the solution of the past experience to
that of the current situation. If an exact match is not found, then the solution to the
previously experienced situation is adapted to fit the existing situation [40].

Knowledge Requirements in Case-Based Systems

Each case is comprised of a “set of empirical data” which has been used in dealing
with a previously experienced situation [42]. A case includes symptoms as well as the
circumstances surrounding the specific situation; a case is a “contextualized piece of
knowledge representing an experience” [40]. Usually a case is decomposed into three
main components: the state of the world when the situation occurred, the solution to the
situation, and the state of the world after the situation occurred [40].

Although the specific content of each case strongly depends on the specific
domain of the application, the stored knowledge in a CBR system should satisfy two
criteria of functionality and ease of retrieving the knowledge [40].

Knowledge Representation and Inferencing in Case-Based Systems

Cases can be formulated by using a number of knowledge representation
methodologies such as frames, object orientation, predicates, semantic nets and rules.
However, the combination of frames and object orientation is the more predominant knowledge representation methodology used in CBR systems.[40]

A few concerns need to be satisfied in formulating and representing the instances and cases. First, to ease the retrieval process, similar cases should be as similar as possible and different cases as dissimilar as possible. Second, the ability to evolve needs to be embedded in the system. This involves being able to classify a new case as close as possible to the case that it is most similar to and farthest from the case that it is most dissimilar to. Third, in order to fill in the blanks for the new cases which are missing values for some of their attributes, the system should be able to refer to the existing cases in the respective class and extract and formulate the values. The fourth concern is with system dependence on the order of instances and cases.[43]

Regardless of how the cases are stored and retrieved, the CBR process could be summarized in four general steps. The situation on hand is evaluated against a similar situation experienced previously. The solution to the previous problem is customized and applied to the existing situation on hand. If applying the customized solution to the problem creates successful or acceptable results, then the solution is remembered the next time in similar circumstances. If, on the other hand, the customized solution is not successful or creates unacceptable results, then the solution to the next closest problem is selected, customized, and applied to the problem on hand. The CBR process is summarized as: recall the most similar cases or episodes, customize the case and apply it to attempt to solve the problem, revise the proposed solution if necessary, and store the new experience including the solution to it as a new case.[40]

Figure 4 displays a suggested system layout, data flow and the main components of a case-based diagnostics expert system.
Discussion on the Third Generation Systems

A CBR system does not always retrieve the closes match. This could be as the result of not having enough cases in the database or not having the right parameters presented in each case to enable the system to distinguish between cases effectively. In the first scenario, the problem is reduced significantly as the system collects more cases. However, to resolve the problem under the second scenario additional parameters or a refined list of parameters might have to be added to each case.[39]

CBR has been found to be a closer match for the human cognitive process than rule-based or model-based methodologies [33, 40].

Figure 4: An example of data flow and main components of a case-based system
2.2.5 Hybrid Diagnostics Expert Systems

Hybrid systems represent a union between shallow and deep knowledge methodologies. The two types of knowledge are integrated in order to take advantage of the whole knowledge continuum as opposed to being limited to the shallow or deep segments. Hybrid systems strategically employ knowledge representation schemes and inference methods of the three methodologies to emulate the human cognitive process. Either of the three approaches in pure format exhibit inefficiencies. Even CBR with its claim of being the methodology closest to the human cognitive process, unless integrated with a model-based system, exhibits limitations in providing effective explanations about the fault on hand [39, 44]. Expert systems presented in [5, 14, 17, 20, 21, 22, 39, 44] take advantage of desirable features of the shallow and deep methodologies and effectively integrate the two approaches to fit the problem domain.

Punch in [14] presents two reasons for his strong inclination towards the integrated approach and why the hybrid approach must be considered instead of deploying shallow or deep approaches in pure format when developing an expert system. First, knowledge is not available all at once and for one purpose only, but instead, it is made available in “piece-meal, at different levels and perhaps for different purposes initially”. Second, any one piece of knowledge “might provide only a part of the answer”. Furthermore, the concepts of “deep” and “shallow” are relative terms [14, 40]. The deep knowledge used in one system could be considered as shallow knowledge in another system. But regardless of whether deep or shallow, knowledge exists on a continuum: the knowledge continuum [21]. The hybrid approach provides a stronger basis (than shallow and deep systems) for modeling the diagnostic action by taking advantage of the knowledge continuum. The diagnostician uses both deep and shallow knowledge to identify and treat a fault: he searches the continuum for the right type of knowledge to apply to the situation when needed [21]. The effectiveness of the approach to developing an expert system, whether based on shallow or deep knowledge, should be primarily based on the constraints and requirements of the problem on hand. “Both deep
and shallow reasoning approaches are effective if applied to problems which justify the use of each approach” [17].

Clearly the three approaches to expert systems have certain advantages and disadvantages. To better examine the integration of the two approaches, it is befitting to examine the merits and demerits of each. A list of criteria for evaluating each methodology along with a summary of evaluation results and brief explanation on each criterion is presented in Table 1.

The degree of integration and whether shallow and deep reasoning are integrated as opposed to linked or interfaced together are determined based on the problem domain. The more common approach to hybrid systems is to interface a rule-based and a model-based system together. Hence, two systems are developed, one with a shallow knowledge-base and another with a deep knowledge-base with two separate inference engines. The two systems share the same user interface module.[20]

Shallower knowledge is used to take a broad look at the system, its components and data, and to produce a suspect list to examine in more detail by using deeper knowledge [19]. “Already known cases of faults can be detected in this manner with much less effort as by causal trouble shooting” [20]. Combining the deep and shallow reasoning effectively enhances the efficiency of the expert system significantly [19]. The causal reasoning system takes over after the fault has been localized by the shallow based system. The model-based inference engine uses “the description of the behaviour and structure of the components and functions in the model to recognize malfunctions” [20]. Recognizing malfunctions in the device, however, is enabled by observing and detecting discrepancies between the observed and expected behaviours, which in turn is the outcome of a simulation run [20].

The hybrid approach cultivates the advantages of both shallow and deep reasoning approaches. Heuristics are used to reduce the size of the model-based which needs to be
examined. The poor runtime of the model-based system due to its simulation requirements is greatly improved when heuristics are used to reduce the size of the functional model. The shallow system provides a limited pre-realized set of faults which potentially could fit the fault on hand; this is quick but limited. The causal model on the other hand, because it depicts the device, accommodates more flexibility but at a slower runtime. The hybrid expert system is "therefore not restricted to a particular realization but has a much wider applicability" [22].
<table>
<thead>
<tr>
<th>Criteria</th>
<th>Rule-Based</th>
<th>Model-Based</th>
<th>Case-Based</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| Best Match for Human Cognitive Process       | Demerit    | Merit       | Merit      | 1. It is a deep understanding of the system architecture that helps diagnosticians in identifying and explaining fault, not just a set of symptomatic responses [17].  
2. However, “human intelligence is approximately 99% pattern recognition and 1% reasoning” [40] |
| Knowledge Modeling and Representation         | Merit      | Demerit     | Merit      | 1. Knowledge representation, depending on the system complexity, could be a major undertaking for model-based systems. [17, 21]  
2. Case-based systems work with empirical systems therefore easier to model [42]  
3. CBR requires less knowledge engineering than model-based systems [33] |
| Development Cost                             | Demerit    | Merit       | Merit      | 1. Potentially low, depending on the availability of the expert whenever needed and also the availability of other sources of knowledge such as diagnostic logs, incident reports, etc.  
2. CBR eliminates the knowledge elicitation bottleneck [40] |
| Development Period                           | Merit      | Demerit     | Merit      | 1. Takes a long time to build a functional model base  
2. CBR can be built without having a functional model [40] |
| Source of knowledge                          | Demerit    | Merit       | Merit      | 1. Agreeing on an “expert” in the field  
2. Unavailability of an expert [2, 27]  
3. Reliance on expert is reduced when gathering cases for CBR [40] |
| Knowledge Acquisition                        | Demerit    | Merit       | Merit      | 1. Could take too long for rule-based due to unavailability of an expert  
2. Lack of familiarity of the knowledge engineer with the domain to ask the right questions [2]  
3. Capturing and formulating expertise knowledge is a major undertaking and challenge [27]  
4. Gathering knowledge for CBR could be as difficult as for model-based [33] |
| Handling New Faults                          | Demerit    | Merit       | Merit      | 1. Rule-based systems are mainly limited to the hard coded rules in the knowledge-base  
2. Model-based systems exhibit flexibility because of causal relationships  
3. CBR can adapt an old solution to the current problem [33, 39, 40] |
<table>
<thead>
<tr>
<th>Criteria</th>
<th>Rule-Based</th>
<th>Model-Based</th>
<th>Case-Based</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| System Complexity                | Merit      | Demerit     | Merit      | 1. Model-based systems tend to have a more complex architecture themselves and they are more often employed for developing expert systems for more complex devices.  
2. Due to short development time, it could be concluded that CBR is less complex than MBR [40] |
| Ease of Modifications            | Demerit    | Merit       | Merit      | 1. Modifying the rule-base is delicate since new rules might clash with old ones [33]  
2. Structural changes could be easily implemented in a model-based system without reviewing the diagnostic procedures [21]  
3. "A CBR system can grow as it gains experience of more cases" [33] |
| Explanation Facilities           | Demerit    | Merit       | Demerit    | 1. Due to their deeper knowledge and superior inferring capabilities, model-based systems could provide a more comprehensive explanations [21]  
2. CBR provide real explanations as opposed to retracing back the steps it took [33] |
| Runtime Efficiency               | Merit      | Demerit     | Merit      | 1. Model-based systems need more processing resources [21]  
2. CBR systems avoid the processing the functional model; CBR is faster than model-based [40] |
| Reusability of Knowledge         | Demerit    | Merit       | Merit      | 1. Little knowledge of the rule-based could be applied to diagnosing an evolved product.  
2. With minor adjustment, model-based systems could be applied to diagnosis of products in the same family [21]  
3. Although no reference was found in literature, it would be reasonable to state that changing case parameters is much more effortless than changing a functional model. |
| Used for other than Diagnostic Purposes | Demerit  | Merit     | Merit      | 1. Model-based diagnostic systems are well suited for training, design validation, FMEA Studies. [21]  
2. Lockheed is using CLAVIER (a case-based system) for multi-purposes [40] |
| System Complexity                | Merit      | Demerit     | Merit      | 1. Model-Based systems take longer and are more complex [21]  
2. CBR systems have a simple architecture compared to model-based systems [40] |
Merits and demerits of the three approaches to expert systems are listed in Table 1. As the number of merits of the case-based approach is greater than that of the other two approaches, the case-based approach seems to be more versatile than rule-based or model-based. However, case-based reasoning is still based on empirical knowledge and as such it inherits the inflexible characteristics of rule-based systems. On the other hand, model-based systems provide the flexibility in diagnosing first-time faults but require significantly more knowledge engineering efforts and resources. Whether one approach is more advantageous over others would also depend on the application and its domain.

2.3 Adopting Strategies for Diagnostics and Expert Systems for This Thesis

The diagnostics strategy selected to support the objectives in this thesis is the model-based reasoning approach. The available information during the design phase of a device or product is mostly of the functional and behavioural type. The design parameters and operating criteria for the device define its internal operations as well as the device's behaviour towards external events or stimuli. The diagnostics strategy should take advantage of the available information during the design phase.

To satisfy the requirements for a diagnostics strategy and the objectives of the thesis, the following definition for the diagnostics process is selected from the literature: "formulation and investigation of hypotheses about the malfunctioning device" [18]. This definition would satisfy the diagnostics process at both ends of the knowledge spectrum: deep as well as shallow. A hypotheses could be formulated and investigated in an attempt to respond to the fault symptoms. Another hypotheses could be formulated and investigated to determine the deeper cause of the fault.
2.4 Search for Similar Research

In the course of the literature review, three separate research works were found to have various degrees of similarity to this project. The first is a methodology for “Generating Fault Hypothesis With a Functional Model in Machine-Fault Diagnosis” by Bublin and Kashyap in [18]. The second is the methodology proposed by Storr and Wiedmann in [20] for creating “An Expert System Shell for Technical Diagnosis”. The third research work presents another expert shell for fault diagnosis by Krishnamurthi and Phillips in [17]. The three research works bear similarities from the knowledge representation and knowledge integration point of view. All three models present methodologies for creating a functional model of a device. They all apply concepts from the Object Orientation domain to the field of model-based reasoning for constructing the computer model of a physical device.

In their proposed methodology for generating fault hypotheses, Bublin and Kashyap propose analyzing a physical component from the logical and physical point of view. Components in a device are organized in different levels of detail and abstraction with assigned behaviours, functions, and products. A component in the device would change, utilize, combine, or separate its inputs to generate outputs. An abnormal behaviour of a component would be attributed to its failure to perform its function or to its inputs. Products flow between components from “source” to “sink” through “gate” and “impetus” functions. Diagnosing faults is performed by detecting a component’s deviation in its normal behaviour and the direction of deviation from the normal.

Storr and Wiedmann present the analysis behind developing an expert system shell for diagnosing technical faults. They present a methodology for integrating the rule-based and model-based diagnostics approaches by including two types of inference mechanisms to accommodate the two types of knowledge: shallow and deep. They propose a hybrid knowledge representation to represent the shallow and deep knowledge. The knowledge base would be comprised of three components: functions, components,
and symptoms, hierarchically organized to depict the physical device. Diagnosing faults is performed by comparing the expected and actual values of outputs. After discovering a discrepancy, the empirical inferencing starts and should it fail to resolve the fault the causal inferencing would follow. The two inference mechanisms interface automatically. The resulting expert shell by Storr and Wiedmann is called DESIS. It was implemented in COOL, an OO language.

The third research work which bears some similarity to this thesis is presented by Krishnamurthi and Phillips in [17]. The system developed in [17] is more than a simple expert shell. It is a system for creating customized expert system for fault diagnosis of electromechanical devices. However, the methodology proposed in [17] could be applied to developing such a system for generating customized expert systems in any domain. The physical model proposed in [17] is comprised of components, materials (flowing between components) and conduits (passive mechanisms for the flow of materials). The expert system generated by the shell is capable of reasoning based on shallow and deep knowledge. The diagnostics process begins with the shallow reasoning and if the symptom is not remedied the diagnostics process proceeds to attempt to remedy the fault by deep reasoning.

The main difference between the work proposed in this thesis and those of the three research projects reviewed above is the degree of integration with the manufacturing environment proposed here. In all three approaches there is some degree of recreating the computer model of the device apart from the design phase. Even in [17] the model-base is recreated for every device. By integrating the development of the model-base with the design phase, greater degree of reuse and sharing of knowledge is achieved and the steps involved in recreating the mode-base could be eliminated altogether.
3. Methodology

The intention is to significantly shorten the time and resources that it takes to design and develop a diagnostics expert system for a device. The proposed methodology will focus on effectively capturing and sharing the device’s functional and behavioural information for designing the device as well as diagnosing its potential faults. The CAD model of a device stores its physical aspects whereas the functional model of the device stores its behavioural aspects.

3.1 Contributing Notions

Four notions predominantly contributed to the formation of the proposed methodology. The first is viewing manufacturing as “a process of embodying useful information in materials”[45]. The second is the notion of parameterizing the attributes and features of objects; then designing a new object could be accomplished by inserting a standard pre-designed object into the model and assigning values to its attributes. The third is causal (or model-based) reasoning in conducting fault diagnostics [18]. The forth notion views entities in this world as objects and the events as the interaction between objects [38].

3.1.1 Manufacturing as Embedding Information into Material

The first notion suggests a new definition for manufacturing from an information perspective befitting of the information age. It views the material undergoing
manufacturing processes as embodiment of information and all the (human) activities in manufacturing as processing that information. To effectively present this notion, as suggested in [45], information embedded in materials and objects around us is divided into two categories: "thermodynamic" and "morphological" information. Thermodynamic information constitute the inherent and innate characteristics of the matter, whereas morphological information describes the geometrical attributes and appearance of the matter [45].

3.1.2 Feature Based or Parametric Design

As the second contributing notion towards this project, parametric or feature based design (FBD) takes advantage of the morphological information in an object for modeling the useful information about the object. Under FBD, the CAD model of a product is created by adding pre-defined features to the model and by assigning values to the parameters or attributes of each feature. FBD enhances productivity by making it easier to change the design in response to changing requirements. Parameterizing the morphological information about an object is the main concept in parametric or feature based design.[11, 36]

3.1.3 Model-Based Diagnostics

The notion of model-based diagnostics in expert systems advocates diagnosing faults based on the deviated behavior of a device from its expected and normal behavior. The more traditional approaches view diagnostics as responding (based on experience) to emerged symptoms. The second generation of diagnostic expert systems probe causal and functional relationships in a device to diagnose its faults. The vital component of a model-based diagnostics expert system is the functional model of a device. The model depicts the device’s normal (designed and therefore, expected) behavior. By introducing different values and propagating the responses throughout the model, one could simulate
the device's behavior. By tracing back the simulated behavior, one could detect the cause(s) of a fault and hence propose effective corrective measures. [18]

3.1.4 Object Orientation

The fourth notion could be considered as the bedrock of implementing the second and third notions. It attempts to define the state of the world as the interaction between individual entities called objects, hence the name of this approach to analysis and design is Object Oriented Methodology. Each object holds its own data and instructions on how to process the data. The object's state is determined by the values it holds for each of its attributes; the object's behaviour is based on how it processes its attribute values. Under this methodology, objects interact by exchanging messages. Under the object oriented methodology, different versions of each entity could exist; therefore, the state of each entity at different instances could be captured and maintained.[48, 49]

3.2 Parameterized-Objects Diagnostics Expert System (PODES)

The four notions are integrated to create virtual models (objects) of physical components. The virtual models would carry information as well as information processing procedures about the component and its immediate surrounding environment. The intelligent entities or virtual models would be able to conduct their own affairs upon receiving control from another object and return the control and reply back to the original object upon completion of what they were asked to do by the original object. Instead of compiling the facts and logical procedures into central structures of the knowledge base and inference engine, each intelligent entity would carry its own facts and logical procedures. The intelligent entities would act as knowledge sources [49]. The domain knowledge is parameterized into the knowledge sources of that domain.
The individual knowledge sources are plugged into one another to create a virtual knowledge base and a virtual inference engine. Implementing such approach would require envisioning the knowledge base and the inference engine as a collection of knowledge sources which are independent of each other yet function cohesively to deliver the functionality which is expected of the diagnostics expert system. As more components are added to the device, their corresponding computer representations or knowledge sources could be plugged into the knowledge base.

The required knowledge sources for a domain are designed, parameterized, and stored in knowledge source libraries. The designer would only need to select the desired knowledge source to be instantiated into the computer model of the device and assign appropriate values to its parameterized attributes. And conversely, as a component is removed from the device, its computer representation is unplugged from the computer model of the device.

Although the entities are independent and autonomous, yet they are related to one another and call upon one another to perform tasks. Once plugged in, each entity would apply its intelligence to either connect itself to the other entities which it would relate to for functionality or to prompt the designer for establishing such functionality. In either case, the designer decides the type of connection and relationships between the entities in the model.

<table>
<thead>
<tr>
<th>Name of KS</th>
<th>KS Resides In</th>
<th>Origin of KS</th>
<th>Content of KS</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Complaints</td>
<td>Complaint Library</td>
<td>User Input</td>
<td>Fault Symptoms</td>
</tr>
<tr>
<td>Faults</td>
<td>Fault Library</td>
<td>Product Design</td>
<td>Fault explanations and conditions</td>
</tr>
<tr>
<td>Relationships</td>
<td>Relationship Library</td>
<td>Product Design</td>
<td>Nature of relationships</td>
</tr>
<tr>
<td>Parts / Components</td>
<td>Part Library</td>
<td>Product Design</td>
<td>Each part holds:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1. attributes which the designer assigns value to during the design process</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2. fault pointers to its potential faults</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3. relationship pointers to faults or relationships and other components</td>
</tr>
</tbody>
</table>
In addition to the physical components of a device, the relationships between its components as well as the potential faults could be considered and modeled as knowledge sources. Table 2 suggests four potential types of libraries and the type of virtual models to be stored in them. The designer would instantiate the knowledge sources as objects into the computer model of the device and establish the required relationships between them by instantiating the required relationship entities.

3.2.1 Independent and Intelligent Component Classes

Each of the pre-designed physical entities is referred to as a component class. The class of each component, relationship, and fault is designed and stored in the component class library. Each class provides the interface to the data and operations of the component, relationship, or fault that it represents. The user’s selection is limited to the components which reside in the component library. Only the components which reside in the component library could be instantiated into the device’s computer model. Each component class is designed by the knowledge engineer and compiled into the component library. By instantiating a component class, the designer is provided with access to the class interface for that object. The class interface would allow the designer to change the parameter values as well as making the functions of that component accessible to the rest of the components in the model.
Each component class is designed to perform a specific task or function. The example of a component class could be the 2" screw component class or a double stroke single cylinder motorcycle engine. The break down of the components would depend on the application and how the component is to be used in the design process. Both the 2" screw and the small engine perform specific tasks; the screw is used for fastening and the engine is used to generate mechanical forces and motion. Whether the engine needs to be represented as one entity or a collection of its constituent parts depends on how the user intends to create the device’s computer model.

Figure 5 shows the parameterized attributes of the domain knowledge which each component object would retain for its operations. Each component class establishes relationships and points of contact by using its pointers, shown in Figure 5 with arrows. Once the user instantiates a component, an object is created in the device’s computer model. Each component object, as shown in Figure 5, holds the knowledge that it needs
regarding its physical as well as operational requirements that it might need to perform on its attribute values in order to change its state.

3.2.2 Relationships

The components of a device could be connected to one another through a variety of relationships such as functional or behavioural, faults, physical or geometrical, as shown in Figure 6. A positional relationship describes how component objects are geometrically positioned relative to one another or relative to an absolute point of reference. Functional and behavioural relationships between two component objects establish operational dependencies between them which in turn establish the operational aspects of the device. A relationship could link two components physically to logically. Physical relationships represent the geometrical aspects of the device and the logical relationships define the operational aspects of the device.

A logical relationship is the modeling representation of one object’s operation on another. A relationship between two components is an operation where one component performs on or requests from the other component. The operation is one of the member functions of a component object in the relationship. The operation is comprised of one or more events.

Based on the object orientation methodology, relationships between component objects could be defined as independent knowledge sources. Just like the component classes, the class of each relationship is designed, compiled, and stored in the relationship library. Therefore, the types of relationships possible between objects is limited to the types of relationship classes stored in the relationship library. The user only instantiates a relationship class once that relationship needs to be established between two component objects. The intelligence for establishing the required relationships could also be embedded in each component class so that upon instantiating a component class, the
component object automatically establishes the relationship with the required component object by instantiating the necessary relationship class.

![Diagram](image)

**Figure 6: Four Examples of Relationships Between Objects**

As shown in Figure 6, two objects could be related through more than one relationship. However, a component object could establish a relationship with another component object only if both component objects have the pointers for that kind of relationship embedded in them: not every component object is able to enter into any relationship. For example, in a hair dryer, the switch is in an On/Off relationship with the fan. The cord is in a Connect/Feed relationship with the switch. But the fan cannot have a Connect/Feed relationship with the cord.

The nature of a relationship dictates the interactions between the objects involved in that relationship, whereas the component objects, through their pointers and embedded functional knowledge, determine when to invoke the operations of their relationship objects. Each relationship object provides certain operations which are invoked upon the receipt of certain messages from the component objects which it relates together. To invoke the operations in the relationship object, the component object needs the pre-designed pointers of the same type as the relationship.

Figure 7 displays the instantiation of component and relationship classes into component and relationship objects to create a network representing the computer model.
of a device. The objects are identified by circles and their respective classes are identified by rectangles. Each relationship object keeps track of the component objects that it is connecting and their types. And in turn each component object keeps track of the relationships it is capable of and the ones that it is using to connect to other component objects. This information will be useful in diagnosing faults which would arise between component objects. By probing the state of the relationship and those of the component objects that it connects together the suspect relationship object and its component objects could be located.

As it is shown in Figure 7, objects interact with one another through relationships. The object could invoke another object’s functions only through relationship objects. Should a component object desire to establish a relationship with another object, it invokes the type of relationship object to instigate the other object on whether it is in a state to establish that kind of relationship with the first component. As soon as the relationship is established, the relationship object stores the two objects as two parties in the relationship and each of the objects are invoked to store the type of relationship with each other.

![Network of Independent and Intelligent Objects](image)

Figure 7: Network of Independent and Intelligent Objects
3.2.3 Faults

As suggested in Figure 8, the contributing factors could be used to segregate the pre-fault environment from the post-fault environment. The pre or post-fault environment is created due to the appearance of certain states in the device which further contribute towards the occurrence of a certain event which would cause the fault to take place. The states could be associated to certain physical components in the device. Furthermore, the contributing components could be associated to a logical system in the device. Therefore, the fault could be related backward to the contributing condition(s), cause(s), system(s), and device(s).

The same analysis could be applied to the post-fault environment. The fault impacts the operating circumstances of some physical components and logical systems.

Figure 8: Suggested Information Flow Pre and Post-Fault
The necessary conditions which need to be present for the fault to occur represent the device's state in the pre-fault environment, whereas the symptoms manifested by the fault after its occurrence represent the device's state in the post-fault environment. Cause of a fault is the event that initiates the state transformation from the initial state (condition) and effect is the event that completes the transformation of the state to the fault state (symptom). The cause works with the pre-fault conditions and the effect generates the post-fault symptoms. The associated events with a fault are associated with the physical components of the device when the component classes are designed.

The events or actions in a device originate from one component and are directed towards another component. For behavioural modeling purposes, it could also be considered that a component could initiate an event or action towards itself. Figure 9 shows the source and destination of reactional dependencies between the physical components of a device. Each component reacts to the actions or events directed towards it by other components or by itself. The reactional dependencies between components manifest the behavioural and functional dependencies between them.

![Diagram of Reactional Dependencies Between Components](image)

**Figure 9: Reactional Dependencies Between Components**

The concept of entity relationships could be extended to establishing fault relationships between component objects. A fault attributed to the link between two objects is initiated and propagated through the device when the reactional dependency between the components is breached. One of the components does not react to an event
or action as expected and its behaviour deviates from the expected or normal behaviour. This could be detected by interrogating the reactional dependencies between components. Each component and relationship object would store the relevant information about the fault at the time of the incident.

As suggested in Figure 9, component objects could be modeled as actors, servers, or agents [49]. Actors initiate action towards other objects; they could be considered as active objects. As the name suggests, servers provide service to their clients, namely, actors. Servers are passive objects. Agents are objects which could be both actors and servers. The role of a component in a relationship could be assigned either manually or automatically. As a part of establishing a link between two objects, the designer of the computer model assigns the actor or server roles to the objects linked together. On the other hand, when an object is instantiated and linked to another object, the link could automatically assign the role of each component. For example, the switch and fan in a hair dryer are linked together with an On/Off relationship. The On/Off relationship could automatically assign the role of actor to the switch and agent to the fan. The role of a component is related to its contribution to the relationship. The role is considered as an attribute value and could be modified by the user. However, since it is tied to the contribution of the component to the relationship, the nature of the link determines if the role of an object in a relationship could be changed. For example, the fan could not turn the switch on; it could not be considered as the active object in its relationship with the switch.

3.3 Fault Diagnostics

Complaint, in the context of fault diagnostics, is defined as a statement about some undesirable physical symptom(s) of a device. The known complaints about the device could be created and stored in the complaint library. A complaint that has not
been experienced yet and is unknown to the designers of the device is created when it is reported by the end user.

![Fault Tree Diagram]

Figure 10: Proposed Fault-Tree for Device Fault Diagnostics

Fault diagnostics under the proposed methodology is conducted by structuring the complaints, faults, links and parts in the proposed fault tree presented in Figure 10. The diamond symbol is used to indicate aggregation, as suggested in [48].

A complaint is an aggregation of the relevant faults. A fault is the aggregation of the breach of at least one reactional dependency. The breach of a reactional dependency is represented by $\neg Link$ in Figure 10. $\neg Link$ refers to the operations in the relationship object which deal with abnormal behaviour(s) of the relationship. A relationship or reactional dependency points to at least two objects. If the two objects are the same it means that the object is performing an internal operation because the same object is the source and sink of the information flow. As it is proposed in Figure 10, the search path for the potential location for the cause of each complaint is identified by aggregating the
relevant faults under a complaint, aggregating the -\textit{links} under faults, and components under the -\textit{links}.

Once the potential problematic relationship is identified, the link performs diagnostics operations to test the validity of its state and that of the component objects that it is linking. While performing diagnostics on itself, the link stores information on the potential faults and reports back that information to the diagnostician. By comparing the reported information to the current state of the device the diagnostician is guided for the next steps in the diagnostics action.

3.4 System Controls

Since the focus of this thesis was stated as parameterizing and capturing diagnostics information during design, minimal attention is dedicated to developing a full scale expert system with advanced system controls. In the proposed methodology it is assumed that the designer or the user of the expert system would determine what needs to be done next when using the system. The user of the expert system would identify the complaint, select the faults to be decomposed into links, and select the desired links for testing. It is the system user who would start with the complaint and find its way down to the component causing the fault. The expert system would only extract and display the required information from the functional knowledge-base. In this capacity, the system could be viewed as diagnostics assistant system as opposed to diagnostics expert system.

3.5 System Architecture

Figure 11 shows a high level system architecture for the proposed methodology. Through the user interface, the user can select the component class to be incorporated into the model from the standard component library. The user also selects the links to be established between the new object and the existing component objects in the model.
Fault objects are inserted along with the component objects. The user does not determine the selection of faults that are attached to each component or relationship object. The system makes this determination. The user can change the attribute values of the fault objects, but not the associativity between the relationship and component and their respective fault objects.

Figure 11: High Level System Architecture for PODES
In Figure 11, the relationship objects are designated by the letter $R$, and the fault objects are designated by the letter $F$. Both the component and the relationship objects are capable of accessing their respective fault objects. When a fault is attributed to the operation of one object onto another, the link between the two objects would need access a link-fault object. When the fault is attributed to the operation of one object on itself, the component object points to a component-fault object.
4. A Prototype System

For the purpose of demonstrating the effectiveness of the proposed methodology a prototype system was developed for a generic electric hair dryer. The developed system allows the user to develop a functional model for a hair dryer, and use the model to diagnose a few potential faults attributed to user complaints such as No Air, No Heat, and No Power which are considered and modeled in this prototype. The underlying assumption for the prototype was that the diagnostician would drive the developed system and decide the next steps. The intent behind the prototype was to focus on the main system functionalities directly associated with the proposed methodology in the thesis, as opposed to developing an extensive model-based diagnostics expert system.

The prototype was implemented in Borland C++ v4.0 for Microsoft Windows 3.1. The proposed methodology is intended to be implemented under OO techniques and environment. C++ is one of the most commercially available OO packages. C++ is not a pure OO programming language; it is the OO extension of C. Since it exhibits the characteristics of structural and OO programming languages, it is called a hybrid language. And it is used in that capacity for developing this prototype.

Rumbaugh’s object oriented methodology was used in the analysis section of this prototype and Booch’s OO methodology was applied to the design and implementation. The choice of OO tools for analysis, design, and implementation was made after reviewing both Rumbaugh’s and Booch’s methodologies. Although they both present their respective methodologies for analysis and design, Rumbaugh presents a stronger focus on object analysis and modeling and has a more structured approach for analysis,
whereas Booch focuses more on the tools and techniques for implementing the OO
concepts.

The database for the prototype is comprised of a number of pointer arrays used for
referencing the objects in the model. To maintain the simplicity of the prototype, no
structured database was deployed to store the functional model-base. Demonstrating the
prototype’s functionality, therefore, would be limited to live data, as opposed to
manipulating a pre-stored functional knowledge-base.

4.1 Physical and Logical System Break Down

As suggested in Bublin and Kashyap’s model, the physical and logical viewpoints
are shown in Figures 12 and 13, respectively. As it is shown in Figure 12, the hair dryer
is an aggregation of fan, heating element, power cord, safety breaker or fuse, and the
on/off switch. The power cord is an aggregation of a cable and a plug. As shown in
Figure 13, the hair dryer could be represented as a collection of systems: a group of parts
which deliver a specific task. Both the physical and logical representations are eventually
broken down into the individual physical components at the lowest layer. A component
could be logically linked to more than one system while physically it is a part of one sub-
assembly.

In this section, Rumbaugh’s Object Modeling Technology is used as a guideline
to identify the functional and behavioural characteristics of the hair dryer.
Figure 12: Physical Representation for Hair Dryer

Figure 13: Logical Representation for Hair Dryer
4.2 Component Classes

The first step towards creating the functional model of the device is identifying and analyzing the classes. Table 3 presents the component analysis for the components of the hair dryer considered in the prototype system.

<table>
<thead>
<tr>
<th>Component Class</th>
<th>Class Attribute</th>
<th>Class Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breaker</td>
<td>• Electric Rating</td>
<td>Disconnect power if circuit is shorted</td>
</tr>
<tr>
<td></td>
<td>• Manufacturer</td>
<td>Ensure safe device operating environment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Invoke no-power fault if flipped</td>
</tr>
<tr>
<td>Fan</td>
<td>• Electric Rating</td>
<td>Suck air from the back of the air duct;</td>
</tr>
<tr>
<td></td>
<td>• Radius</td>
<td>Blow air over the heating element through the air</td>
</tr>
<tr>
<td></td>
<td>• Shape</td>
<td>nozzle in the forward direction.</td>
</tr>
<tr>
<td></td>
<td>• Type of material</td>
<td>Invoke no-air fault if not turning</td>
</tr>
<tr>
<td></td>
<td>• Speed</td>
<td>Invoke no-air fault if not connected to switch</td>
</tr>
<tr>
<td>Element</td>
<td>• Electric Rating</td>
<td>Convert electricity to heat under three</td>
</tr>
<tr>
<td></td>
<td>• Length</td>
<td>automatically selected levels of low, medium, and high.</td>
</tr>
<tr>
<td></td>
<td>• Shape</td>
<td>Invoke no-heat-fault if not working and switch is in</td>
</tr>
<tr>
<td></td>
<td>• Type of material</td>
<td>fan-heat position</td>
</tr>
<tr>
<td>Cord</td>
<td>• Electric Rating</td>
<td>Invoke no-heat-fault if element is broken and switch</td>
</tr>
<tr>
<td></td>
<td>• Length</td>
<td>is in fan-heat position</td>
</tr>
<tr>
<td></td>
<td>• Kind (grounded or not)</td>
<td>Invoke no-power if not connected to breaker</td>
</tr>
<tr>
<td>Switch</td>
<td>• Electric Rating</td>
<td>Facilitate device operations through user selections;</td>
</tr>
<tr>
<td></td>
<td>• Number of Positions</td>
<td>provides 3 selections:</td>
</tr>
<tr>
<td></td>
<td>(3)</td>
<td>Off: no power</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A: Only fan on (air only)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B: Both fan and heater on (hot air)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Invoke bad fan-contact fault</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Invoke bad element-contact fault</td>
</tr>
<tr>
<td>Hair Dryer</td>
<td>• Electric Rating</td>
<td>A device for drying hair by blowing air. User can select</td>
</tr>
<tr>
<td></td>
<td>• Colour</td>
<td>cold or hot air</td>
</tr>
<tr>
<td></td>
<td>• Weight</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Operating Environment</td>
<td></td>
</tr>
</tbody>
</table>

There are common features carried by all component classes. Each component class store information on the components that it is in relationship with. Each component
also stores information on when and how to invoke any fault(s) it contributes to, as well as information on steps toward removing the fault condition. Relationship classes, on the other hand, store information on how to handle fault invocations from the component classes. Furthermore, a relationship class should have the knowledge on how to pass fault conditions between the component classes that it links together. Table 4 shows three types of relationship classes used in modeling the hair dryer.

<table>
<thead>
<tr>
<th>Relationship Class</th>
<th>Class Attribute</th>
<th>Class Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect / Feed</td>
<td>Connected components, Means of connection</td>
<td>Link two components electrically; Keep track of what components are connected to one another; Assign direction for the connection Knowledge of what fault to invoke under undesirable conditions</td>
</tr>
<tr>
<td>Blow Over</td>
<td>Air volume per second Blower Heater</td>
<td>Blow air to remove excess heat and hence reduce temperature</td>
</tr>
<tr>
<td>On/Off Control</td>
<td>Name of On component Name of Off component</td>
<td>Turn on the fan Turn on the element (whether fan is on or off)</td>
</tr>
</tbody>
</table>

4.3 Component Object Model of Hair Dryer

The object model integrates the component and relationship classes. It graphically represents the component classes linked together through relationship classes. Figure 14 displays a suggested object model for the hair dryer. The component object model depicts the static aspects of a device. It shows what components exist in a device and how they are related to one another.
In Figure 14, relationships are specified by the text adjacent to the line connecting the component objects. Whether to define all relationships as classes is a design decision; however, defining all the relationships as classes would raise the consistency of the device model.

4.4 Dynamic Model of Hair Dryer

The main purpose of the dynamic model is to depict the flow of control through the system. It shows how a device responds to outside stimuli. The dynamic model is simply a collection of state diagrams showing state transition of the classes in the model. Figure 15 shows the state diagrams for the fan and the element. A state corresponds to a period of time whereas an event corresponds to a point in time [48, 49]. In Figure 15, the rounded rectangles show different states of the hair dryer or of its components and the text adjacent to the links specify the event which needs to take place to transform one state to the next.
4.5 Fault Analysis for Hair Dryer

No specific step is dedicated to fault analysis in OMT. In fact, events which transfer the device to a fault condition are no different from any other event in the system. However, due to the fact that they generate undesirable results they are grouped together and referred to as fault events and conditions. Faults could be modeled as classes just like components and relationships. Table 5 shows the two classes of faults implemented in this prototype.

<table>
<thead>
<tr>
<th>Fault Class</th>
<th>Class Attribute</th>
<th>Class Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Heat</td>
<td>Heat Source, Heat Source Power</td>
<td>Investigate the element for its condition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Investigate the element’s source of power</td>
</tr>
<tr>
<td>No Air</td>
<td>Air Source, Air Duct, Air Source</td>
<td>Investigate the fan for its condition</td>
</tr>
<tr>
<td></td>
<td>Power</td>
<td>Investigate the fan’s power</td>
</tr>
</tbody>
</table>

Table 6 shows the device condition from which the fault is initiated and the symptoms for the fault condition as well as the events which take place to create the fault
condition and the events which do not happen as the result of the fault condition. In this prototype the symptoms are the same as the complaints which the user would express once the fault is experienced.

<table>
<thead>
<tr>
<th>Fault</th>
<th>Condition</th>
<th>Cause</th>
<th>Effect</th>
<th>Symptom</th>
</tr>
</thead>
</table>
| No Heat | Switch is in Pos. 3 (Fan is working) | - Broken element  
- No open electric circuit | No heat generated | - The device blows cold air whether the switch is in position 2 or 3. |
| No Heat | Switch is in Pos. 3 (Fan is working) | - Bad or loose connection for the element at the switch or element | No heat generated | - The device blows cold air whether the switch is in position 2 or 3. |
| No Air | Switch is in Pos. 2 | - Bad or loose connection for the element at the switch  
- Breaker off  
- Broken Fan | Fan is not turning | - No air flow |
| No Air | Switch is in Pos. 3 (element heats up) | - Broken Fan | Fan is not turning | - No air flow |

Faults could also be associated to physical components in the device. Table 7 shows what components are involved in initiating each of the two faults under consideration in this prototype and which components in the device are impacted. It is also important to consider the logical systems which are involved in the fault, either initiating or effected.

<table>
<thead>
<tr>
<th>Fault</th>
<th>Initiating System</th>
<th>Initiating Part</th>
<th>Affected Part</th>
<th>Affected System</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Heat</td>
<td>Heating system</td>
<td>Element</td>
<td>Element</td>
<td>Heating System</td>
</tr>
<tr>
<td>No Heat</td>
<td>Power System</td>
<td>Switch</td>
<td>Element</td>
<td>Heating System</td>
</tr>
<tr>
<td>No Air</td>
<td>Power System</td>
<td>All Power System components</td>
<td>Fan, Element</td>
<td>Heating System, Air System</td>
</tr>
<tr>
<td>No Air</td>
<td>Air System</td>
<td>Fan</td>
<td>Fan, Element</td>
<td>Heating System, Air System</td>
</tr>
</tbody>
</table>
4.6 Functional Model of Hair Dryer

The functional model represents the operations in the device: how things happen. Where the dynamic model represents a collection of state diagrams for the device, the functional model is made up of multiple data flow diagrams. Where the object model shows the components involved in the system and the dynamic model shows the timing and sequence of operations, the functional model shows how the operations take place. The Dynamic model shows the state transformations whereas the functional model shows the operations transformation.

4.7 Implementation Issues

The designer of the functional model should satisfy two requirements prior to embarking on the task of designing the model. First, the designer should have the background and familiarity with the domain knowledge. Much of the analysis work presented in this chapter needs to be done prior to developing the model. By going through the analysis phase, the designer would become familiar with the device’s operations. Prior to designing the functional model for a device, the designer should become intimately familiar with the devices operations. Availability of standard component and relationship classes could be a guide for the designer as to what to do next. But the underlying assumption here is that the designer has the background and familiarity with the device and domain knowledge.

Second, the designer should be able to logically link the operations of the components of the device. Prior to linking the component objects together, the designer should know about the operations that each component insert on other objects in that device. Without such information, the links in the model might not reflect the real world operations of the device.
5. Concluding Discussion

The research conducted in this thesis exhibits advantages and disadvantages and provides opportunities as well as limitations. Some of which are discussed in the following sections.

5.1 Conclusion

This thesis addressed the main challenges experienced in developing expert systems, namely the challenge of gathering domain knowledge for the development of the knowledge base. As a secondary activity, this thesis also presented an alternative to the central inferencing and knowledge base strategy. A comprehensive literature review was conducted to capture the state of the art in expert systems for diagnostics and to identify supporting research projects for the two objectives of this thesis.

Three similar researches were located during the literature review: Bublin and Kashyap in [18], Krishnamurthi and Phillips in [17], and Storr and Wiedmann in [20]. However, none consider integrating the development activities for the functional knowledge-base and that of the CAD model.

The objectives of this thesis were: to develop a methodology for parameterizing and capturing the functional knowledge of a device while it is being designed, and to develop a tool which would demonstrate the methodology. Four underlying concepts contributed to the formation of the proposed methodology. The four concepts are:
manufacturing as embedding useful information into material, feature based design, model-based diagnostics, and object orientation it is possible to effectively integrate designing a device and developing its functional model.

Both objectives were demonstrated. The first objective was satisfied by first analyzing the nature of the diagnostics information and then modeling that information as attributes in three primary classes of virtual objects, namely component, relationship, and fault classes. The second objective of this thesis was also met by developing a prototype model for a diagnostics expert system for a hair dryer. The prototype was implemented in Borland C++ v4.0 under OOM.

It was demonstrated in this thesis that it is possible to parameterize and capture diagnostics information about a product while it is being designed. Such practice would shorten the development cycle of the expert system. Parameterizing and capturing diagnostics information was also introduced as one of the first steps towards integrating the functional and geometrical knowledge-bases.

5.2 Advantages

Designing based on FBD concepts automatically creates consistency in design. Not only is the design process standardized, but also the computer model of any product would consist of the same attributes and characteristics because all designers would be provided with the same object libraries from which they would pick and add into the model. Hence, designing the functional model in such an environment would be limited to picking the right component and assigning the correct values for the object's parameters. Significant effort and time is saved by eliminating the redesigning and remodeling the components of the functional model for every new system.
5.3 Disadvantages

Standardization would always create opposition amongst those who believe it would dampen their creativity. The designers are limited to the variety of objects included in the standard component, link and fault libraries.

Another disadvantage for implementing the proposed methodology is the effort required for creating the standard libraries in a domain. Developing standard libraries for component objects, links, and test patterns is a monumental task requiring significant resources and efforts. It is not an impossible task but one which would require time and management.

From a computing perspective, the objects carried in a computer model developed by following the methodology proposed in this thesis might carry overhead information which may never be used. The size of the model base would grow significantly if each object would carry its design description as well as its functional and behavioural information, unless the user is provided with the functionality to include or preclude the information used for fault diagnostics.

5.4 Practicality

Parametric design has been accepted as a stable, reliable, and efficient design paradigm [34]. It generates savings by eliminating the need to redesign an object from scratch every time. By designing and parameterizing its attributes once, it could be applied to many models over and over again simply by assigning the desired values to its attributes. Parametric design has been implemented successfully in CAD. A number of object libraries exist for mechanical and electrical design which provide designers with pre-designed objects in a CAD environment which could be simply incorporated into the model being designed. Having standard libraries available for design would save the
designers time and effort. However, the question of whether it would be practical (feasible) to design functional models of all electromechanical components in a domain would require research which is beyond the concerns of this thesis. Such research should examine the economic concerns as well as the perception of a more restricted design environment under FBD.

5.5 Future Possibilities

The total integration of the model base of a model-based diagnostics expert system and the computer model of the device in a CAD environment is yet to be demonstrated. Such an integrated environment would facilitate creating product models which could be used in manufacturing planning, product inspection and quality control, and fault diagnostics. Since the computer model carries functional and behavioural knowledge, it could also be used for user training.

Another opportunity would be to allow the processing of shallow knowledge by adding an empirical inference mechanism and integrating a case-based system to allow the knowledge base to dynamically grow as more faults are resolved by the system.
Appendix A: C++ Code for PODES

/ * Parameterized Objective Diagnostics Expert System - PODES *

* Copy Right: Roozbeh Rahimpour, December 1996 *

******************************************************************************

/ *
* Adding the OWL Header files *
*/

#include <owl\owlpch.h>
#include <owl\applicat.h>
#include <owl\framewin.h>
#include <owl\static.h>
#include <owl\inputdia.h>
#include <owl\validate.h>
#include <string.h>
#include <stdio.h>
#include <owl\button.h>
#include <owl\checkbox.h>
#include <owl\df.h>
#include <owl\listbox.h>
#include <owl\edit.h>
#include <owl\dialog.h>
#include <owl\menu.h>

* 
* Local include file *
*

#include "desmenu.h"
#include "faults1.h"
#include "parts1.h"
#include "links1.h"
#include "complain1.h"
#include "faults2.h"
#include "parts2.h"
#include "links2.h"
#include "complain2.h"

/*
* Local Macros or Define PreProcessor Statements
*/
#define TRUE 1
#define FALSE 0

const WORD ID_BUTTON1 = 101;
//const WORD ID_BUTTON2 = 102;
const WORD ID_BUTTON3 = 103;
const WORD ID_BUTTON4 = 104:

class TTestWindow : public TWindow {
  public:
    TTestWindow();

    // button handlers
    void HandleButton1Msg(); // ID_BUTTON1
    void HandleButton2Msg(); // ID_BUTTON2
    void HandleButton3Msg(); // ID_BUTTON3
    void HandleButton4Msg(); // ID_BUTTON4

    DECLARE_RESPONSE_TABLE(TTestWindow);
};

DEFINE_RESPONSE_TABLE1(TTestWindow, TWindow)
  EV_COMMAND(ID_BUTTON1, HandleButton1Msg),
  // EV_COMMAND(ID_BUTTON2, HandleButton2Msg),
  EV_COMMAND(ID_BUTTON3, HandleButton3Msg),
  EV_COMMAND(ID_BUTTON4, HandleButton4Msg),
END_RESPONSE_TABLE;

TTestWindow::TTestWindow()
  : TWindow(0, 0, 0)
{
  // Define databases for the devices, faults, and relationships

  // DataBase    deviceDB[50];
  // DataBase    faultsDB[5];
  // DataBase    relateDB[5];

  // Define buttons on the main PODES screen

    new TButton(this, ID_BUTTON1, "Design+Diagnostics",
                200, 80, 296, 24, FALSE);
    new TButton(this, ID_BUTTON3, "About PODES", 200, 130, 296, 24, FALSE);
    new TButton(this, ID_BUTTON4, "Exit", 200, 155, 296, 24, FALSE);
}

void TTestWindow::HandleButton3Msg()
{
    MessageBox("Parameterized Objective Diagnostics Expert System\n        \n        Alpha Prototype Version\n        \n        All Rights Reserved,\n        \n        Global Dynamics Systems Research, Inc., April 1997.",
        "About PODES", MB_OK);

```cpp
void TTestWindow::HandleButton4Msg()
{
    exit(0);
}

//--------------- Main PODES Window Application ---------------

class TTestApp : public TApplication {
    public:
        TTestApp() : TApplication() {}
        void InitMainWindow();
};

void TTestApp::InitMainWindow()
{
    MainWindow = new TFrameWindow(0,
    "Parameterized Objective Diagnostics Expert System - PODES",
    new TTestWindow);
}

int OwlMain(int /*argc*/, char* /*argv*/ [])
{
    return TTestApp().Run();
}

/*-------------------------------------------------------------
   *
   *----------------------------------------------------------------
   *------------------ List Dialog Box Definitions ----------------*
   *
   *----------------------------------------------------------------

class TDlgDialog: public TDialog
{
    public:
        TDlgDialog(TWindow* parent, TResID resID, int* n);
        bool EvInitDialog(HWND);
        int* pSelection;
        char sDialogData[25][50];
        void HandleListBoxMsg();
        int GetSelection();

        DECLARE_RESPONSE_TABLE(TDlgDialog);
};

DEFINE_RESPONSE_TABLE1(TDlgDialog, TDialog)
    EV_LBN_SELCHANGE(101, HandleListBoxMsg),
END_RESPONSE_TABLE;

TDlgDialog::TDlgDialog(TWindow* parent, TResID resID, int* n)
    : TDialog(parent, resID), TWindow(parent)
{
    pSelection = n;
}
BOOL TDlgDialog::EvInitDialog(HWND)
{
    for(int i = 0; i < 5; i++)
    {
        SendDlgItemMessage(101, LB_INSERTSTRING, i, (LPARAM) sDialogData[i]);
    }
    return TRUE;
}

void TDlgDialog::HandleListBoxMsg()
/
    *pSelection = (int) SendDlgItemMessage(101, LB_GETCURSEL);
/
int TDlgDialog::GetSelection()
{
    return *pSelection;
}

/*
 *          Edit Dialog Box Definitions ---------------
 */

class TEditDialog: public TDialog
{
    public:
    TEditDialog(TWindow* parent, TResID resID, int* n);
    TEdit* edBox:

    DECLARE_RESPONSE_TABLE(TEditDialog):
};

DEFINE_RESPONSE_TABLE1(TEditDialog, TDialog)
END_RESPONSE_TABLE;

TEditDialog::::TEditDialog(TWindow* parent, TResID resID, int* n)
    : TDialog(parent, resID), TWindow(parent)
{
    edBox = new TEdit(this, 101, 24);
}

/*
 *         Define the Design Application class -------
 */

class DesApp : public TApplication
{
    public:
DesApp( char far *name =0 ):TApplication( name ) {
    virtual void InitMainWindow( void );
};

/*
 * Define the main window class
 */

class DesMainWin : public TFrameWindow
{

public:

    // Define the Parts, Faults, and Relationship Libraries
    char sPartsLib[5][25];
    char sFaultsLib[5][25];
    char sLinksLib[5][25];
    char sComplaintsLib[5][25];

    // Define the Part, Faults, and Relationship Containers
    Part  *vParts[20];
    Fault *vFaults[20];
    Link  *vLinks[20];
    Complaint *vComplaints[20];
    int    linksCount;
    int    partsCount;
    int    faultsCount;
    int    complaintsCount;
    int    nCandidFaultsIndex;
    int    nCandidFaultsCount;

    // Define the output character arrays for the dialog box
    char out_string[25];
    char sListBoxData[25][25];
    int  nListBoxItems;
    int  nListBoxIndex;
    char *pTemp;

DesMainWin( TWindow *parent, const char far *title):
    TFrameWindow(parent, title)
{
    MBoxEnabled = TRUE;
    for (int i = 0; i < 25; i++)
    {
        sListBoxData[i][0] = 0;
    }
    nListBoxItems = 0;
    nListBoxIndex = 0;
    out_string[0] = 0;

    linksCount = 0;
    partsCount = 0;
    faultsCount = 0;
    complaintsCount = 0;
    nCandidFaultsIndex = 0;
nCandidFaultsCount = 0;

// Initialize the Parts Library
strcpy(sPartsLib[0], "Fan");
strcpy(sPartsLib[1], "Element");
strcpy(sPartsLib[2], "Cord");
strcpy(sPartsLib[3], "Switch");
strcpy(sPartsLib[4], "Breaker");

// Initialize the Faults Library
strcpy(sFaultsLib[0], "No Heat");
strcpy(sFaultsLib[1], "No Air");
strcpy(sFaultsLib[2], "No Power");
sFaultsLib[3][0] = 0;
sFaultsLib[4][0] = 0;

// Initialize the Relationships Library
strcpy(sLinksLib[0], "Turn On/Off");
strcpy(sLinksLib[1], " Blow Over");
strcpy(sLinksLib[2], " Blow Through");
strcpy(sLinksLib[3], " Connect");
strcpy(sLinksLib[4], " Feed");

// Initialize the Complaint Library
strcpy(sComplaintsLib[0], "No Heat");
strcpy(sComplaintsLib[1], " No Air");
strcpy(sComplaintsLib[2], " No Power");
sComplaintsLib[3][0] = 0;
sComplaintsLib[4][0] = 0;

// Load up the Complaints Collection Vector

NoHeat* noHeat = new NoHeat;
vComplaints[complaintsCount++] = noHeat;

NoAir* noAir = new NoAir;
vComplaints[complaintsCount++] = noAir;

NoPower* noPower = new NoPower;
vComplaints[complaintsCount++] = noPower;

};

void Paint(TDC &DC, BOOL bErase, TRect &invalidRect);

void CmAddComponent();
void CmDelComponent();
void CmModComponent();
void CmListComponent();
void CmSelectAssembly();
void CmCreateAssembly();
void CmDelAssembly();
void CmModAssembly();
void CmListAssembly();
void CmListAllAssemblies();

void CmSetupRel();
void CmDelRel();
void CmModRel();
void CmListRel();

void CmAddComplaint();
void CmDelComplaint();
void CmModComplaint();
void CmListComplaints();
void CmListFails();

void CmGetComplaint();
void CmDispCause();
void CmDispCondi();
void CmDispInitSys();
void CmDispInitPart();
void CmDispEffect();
void CmDispSymptom();
void CmDispEffSys();
void CmDispEffPart();
void CmDispHowToFix();

void CmDisplayMBoxEnable(TCommandEnabler &tce);

DECLARE_RESPONSE_TABLE(DesMainWin);
private:
    BOOL MBoxEnabled;

/*
 * Define the response table for the menu
 */

DEFINE_RESPONSE_TABLE1( DesMainWin, TFrameWindow )
    EV_COMMAND( CM_FILEEXIT, CmExit ),
    EV_COMMAND( CM_ADDCOMPONENT, CmAddComponent ),
    EV_COMMAND( CM_DELCOMPONENT, CmDelComponent ),
    EV_COMMAND( CM_MODCOMPONENT, CmModComponent ),
    EV_COMMAND( CM_LISTCOMPONENT, CmListComponent ),
    EV_COMMAND( CM_SELECTASSEMBLY, CmSelectAssembly ),
    EV_COMMAND( CM_CREATEASSEMBLY, CmCreateAssembly ),
    EV_COMMAND( CM_DELASSEMBLY, CmDelAssembly ),
    EV_COMMAND( CM_MODASSEMBLY, CmModAssembly ),
    EV_COMMAND( CM_LISTASSEMBLY, CmListAssembly ),
    EV_COMMAND( CM_LISTALLASSEMBLIES, CmListAllAssemblies ),
    EV_COMMAND( CM_SETUPREL, CmSetupRel ),
    EV_COMMAND( CM_DELREL, CmDelRel ),
    EV_COMMAND( CM_MODREL, CmModRel ).
void DesAppl::InitMainWindow(void)
{
    SetMainWindow(new DesMainWin(0, "PODES - Design Module"));
    GetMainWindow()->AssignMenu( "DESMENU" );
}

void DesMainWin::Paint(TDC &tDC, BOOL bErase, TRect &invalidRect)
{
    TRect clientArea;
    GetClientRect(clientArea);
    tDC.SetTextAlign(TA_CENTER);
}

void DesMainWin::CmAddComponent()
{
    int nHere = -1;
    int toDelete;
    char editData[25];
    editData[0] = 0;
    TDlgItem* listDialog = new TDlgItem(this, TResID(1000), &nHere);
    for(int i = 0; i < 5; i++)
    {
        char append_string[25];
        int append_index = 0;
        while(sPartsLib[i][append_index] != ' ' && sPartsLib[i][append_index] != 0)
        {
            append_string[append_index] = sPartsLib[i][append_index++];
        }
append_string[append_index] = 0;
strcpy(listDialog->sDialogData[i], append_string);

if(listDialog->Execute() == IDOK & & nHere != -1)
{
    strcpy(out_string, sPartsLib[nHere]);
nListBoxIndex = strlen(out_string);
Invalidate(TRUE);
}

int selection = listDialog->GetSelection();

if(listDialog->Execute() == IDOK & & nHere != -1)
{
switch (selection)
{
    case 0:
    {
        MessageBox("Adding a Fan", "Your Selection:", MB_OK);
        Fan* fan = new Fan;
        TEditDialog* editBox = new TEditDialog(this, TResID(100), &nHere);
        editBox->SetTransferBuffer(editData);
        if(editBox->Execute() == IDOK )
        {
            for (i = 0; i < 25; i++)
            {
                fan->name[i] = editData[i];
            }
            vParts[partsCount++] = fan;
        }
    }
    BrokenFan* brokenFan = new BrokenFan; 
    for (i = 0; i < faultsCount; i++)
    {
    if(strcmp(vFaults[i]->name, brokenFan->name))
        toDelete = FALSE;
    else
        toDelete = TRUE;
    }
    if (toDelete)
        delete brokenFan;
    else
    {
        brokenFan->initBrokenFanFault();
        vFaults[ faultsCount++] = brokenFan;
    }
    if (0 < faultsCount)
        MessageBox("Adding Fan Fault-Objects", "Your Selection:", MB_OK);
    break;
    }
    case 1:
    {
        MessageBox("Adding a Heating Element", "Your Selection:", MB_OK);
        Element* element = new Element;
        TEditDialog* editBox = new TEditDialog(this, TResID(100), &nHere);
editBox->SetTransferBuffer(editData);
if(editBox->Execute() == IDOK )
{
    for (i = 0; i < 25; i++)
        element->name[i] = editData[i];
}
vParts[partsCount--] = element;

/*
BrokenElement* brokenElement = new BrokenElement;
for (i = 0; i < faultsCount; i++)
{
    if (!strcmp(vFaults[i]->name, brokenElement->name))
        toDelete = FALSE;
    else
        toDelete = TRUE;
}
*/
if (toDelete)
    delete brokenElement;
else
    /*
    brokenElement->initBrokenElementFault();
    vFaults[faultsCount++] = brokenElement;
    */

BadSwitchPos* badSwitchPos = new BadSwitchPos;
/*
for (i = 0; i < faultsCount; i++)
{
    if (!strcmp(vFaults[i]->name, badSwitchPos->name))
        toDelete = FALSE;
    else
        toDelete = TRUE;
}
*/
if (toDelete)
    delete badSwitchPos;
else
    /*
    badSwitchPos->initBadSwitchPosFault();
    vFaults[faultsCount++] = badSwitchPos;
    */

if ( 0 < faultsCount )
    MessageBox("Adding Element Fault-Object", "Your Selection:", MB_OK);
break;
}

case (2):
{
    MessageBox("Adding a Cord", "Your Selection:", MB_OK);
    Cord *cord = new Cord;
    TEditDialog* editBox = new TEditDialog(this, TRestID(100), &nHere);
    editBox->SetTransferBuffer(editData);
    if(editBox->Execute() == IDOK )
    {
        /*
        cord->SetTransferBuffer(cordData);
        */
        MessageBox("Adding a Cord to Fault-Object", "Your Selection:", MB_OK);
        /*
        vFaults[faultsCount++] = cord;
        */
    }
}

if (faultsCount < 1)
for (i = 0; i < 25; i++)
    cord->name[i] = editData[i];
}
vParts[partsCount++] = cord;

BadCord* badCord = new BadCord;
for (i = 0; i < faultsCount; i++)
{
    if (!strcmp(vFaults[i]->name, badCord->name))
        toDelete = FALSE;
    else
        toDelete = TRUE;
}
if (toDelete)
    delete badCord;
else
    { /*
    badCord->initBadCordFault();
    vFaults[faultsCount++] = badCord;
    */

    if (0 < faultsCount)
        MessageBox("Adding a Bad-Cord Fault-Object", "Update Msg:", MB_OK);
    break;
}
}

vParts[partsCount++] = key:

BadSwitchPos* badSwitchPos = new BadSwitchPos;
/*
for (i = 0; i < faultsCount; i++)
{
    if (!strcmp(vFaults[i]->name, badSwitchPos->name))
        toDelete = FALSE;
    else
        toDelete = TRUE;
}
*/

if (toDelete)
    delete badSwitchPos;
else
badSwitchPos->initBadSwitchPosFault();
vFaults[faultsCount++] = badSwitchPos;

//

if( 0 < faultsCount )
    MessageBox( "Adding Bad Switch Fault-Object", "Update Msg: ", MB_OK);
    break;
}

case (4):
{
    MessageBox( "Adding a Fuse", "Your Selection:", MB_OK);
    Breaker *breaker = new Breaker;
    TEditDialog* editBox = new TEditDialog(this, TRestID(100), &nHere);
    editBox->SetTransferBuffer(editData);
    if(editBox->Execute() == IDOK )
    {
        for(i = 0; i < 25; i++)
            breaker->name[i] = editData[i];
    }
    vParts[partsCount++] = breaker;

    BadFuse* badFuse = new BadFuse;
    for(i = 0; i < faultsCount; i++)
    {
        if(!strcmp(vFaults[i]->name, badFuse->name))
            toDelete = FALSE;
        else
            toDelete = TRUE;
    }
    if(toDelete)
        delete badFuse;
    else
    { /*
        badFuse->initBadFuseFault();
        vFaults[faultsCount++] = badFuse;
    */
        break;
    }
}

default:
{
    MessageBox("Bad Selection, Try Again!", "Your Selection:", MB_OK);
    break;
}
}

//Select the assembly from a list to add to
//Select the component to add to the assembly
//assign attribute values to the component
//setup relationships for the component
//setup faults for the component
void DesMainWin::CmDelComponent()
{
    // Get the name of the component to be deleted
    // Ask the user for delete confirmation
    // remove the component from the assembly
    // remove the component's fault relationships
    // remove the component's physical relationships
}

void DesMainWin::CmModComponent()
{
    // Select the component from a list
    // List the component with the values exposed for modification
}

void DesMainWin::CmListComponent()
{
    int i, j;
    char* pCharTemp;

    if (partsCount == 0)
    {
        MessageBox("No parts in the model for listing", "Note!", MB_OK);
    }
    else
    {
        for (j = 0; j < partsCount; j++)
        {
            for (i = 0; i < 25; i++)
                sLBoxData[j][i] = vParts[j]->name[i];
        }
    }

    int nHere = -1;

    TDlgDialog* listDialog = new TDlgDialog(this, TResID(1003), &nHere);
    for(i = 0; i < 25; i++)
    {
        char append_string[25];
        int append_index = 0;
        while(sLBoxData[i][append_index] != ' ' 
              && sLBoxData[i][append_index] != 0)
        {
            append_string[append_index] = sLBoxData[i][append_index++];
        }
        append_string[append_index] = 0;
        strcpy(listDialog->sDialogData[i], append_string);
    }

    if(listDialog->Execute() == IDOK && nHere != -1)
    {
        strcpy(out_string, sLBoxData[nHere]);
        nListBoxIndex = strlen(out_string);
        Invalidate(TRUE);
    }

    int selection = listDialog->GetSelection();

    if(listDialog->Execute() == IDOK && nHere != -1)
{
    switch (selection)
    {
    case (0):
        
            MessageBox("Listing Information Case#0", "Your Selection:", MB_OK);
            break;
        
    case (1):
        
            MessageBox("Listing Information Case#1", "Your Selection:", MB_OK);
            break;
        
    case (2):
        
            MessageBox("Listing Information Case#2", "Your Selection:", MB_OK);
            break;
        
    case (3):
        
            MessageBox("Listing Information Case#3", "Your Selection:", MB_OK);
            break;
        
    case (4):
        
            MessageBox("Listing Information Case#4", "Your Selection:", MB_OK);
            break;
        
    default:
        
            MessageBox("Bad Selection, Try Again!", "Your Selection:", MB_OK);
            break;
    
    } //Select the component from the component list
    //List the attributes and associated values for that component
}

void DesMainWin::CmSelectAssembly()
{
    
    //List all assemblies
    //Select one assembly
    //Commit to memory with the expectation of being further manipulated
}

void DesMainWin::CmCreateAssembly()
{
    TMenu MenuObj(GetMenu() );
}
UINT CurrentMenuState = MenuObj.GetMenuState(CM_CREATEASSEMBLY.
MF_CHECKED);

MenuObj.CheckMenuItem(CM_CREATEASSEMBLY,
CurrentMenuState ? MF_UNCHECKED : MF_CHECKED);

DrawMenuBar();

}

void DesMainWin::CmDelAssembly()
{
   //List all assemblies
   //Select one assembly
   //Delete the selected assembly
}

void DesMainWin::CmModAssembly()
{
}

void DesMainWin::CmListAssembly()
{
}

void DesMainWin::CmListAllAssemblies()
{
}

void DesMainWin::CmSetupRel()
{
   if (partsCount <= 0)
      MessageBox("No Part Objects in the Model!\n"."Warning Message:". MB_OK);
   else
      {
int nCandidPartCount = 0;
int nCandidPartIndex = 0;
int nCandidLinkIndex = 0;

int nHere = -1;
char editData[51];
editData[0] = 0;

TDlgDialog* listDialog = new TDlgDialog(this, TResID(3000), &nHere);
for(int i = 0; i < 5; i++)
{
   char append_string[25];
   int append_index = 0;
   while(sLinksLib[i][append_index] != '
      && sLinksLib[i][append_index] != 0)
      {
append_string[append_index] = sLinksLib[i][append_index++];
}
append_string[append_index] = 0;
strcpy(listDialog->sDialogData[i], append_string);
}
if(listDialog->Execute() == IDOK && nHere != -1)
{
   strcpy(out_string, sLinksLib[nHere]);
}
nLBoxIndex = strlen(out_string);
Invalidate(TRUE);

int selection = listDialog->GetSelection();

if(listDialog->Execute() == IDOK && nHere != -1)
{
    switch (selection)
    {
    case (0):
    {
        MessageBox("Setting Up On/Off Link", "Your Selection: ", MB_OK);
        OnOff *onOff = new OnOff;
        TEditDialog* editBox = new TEditDialog(this, TResID(300), &nHere);
        editBox->SetTransferBuffer(editData);
        if(editBox->Execute() == IDOK )
        {
            for (i = 0; i<50; i++)
                onOff->name[i] = editData[i];
        }
        vLinks[linksCount++] = onOff;
        nCandidLinkIndex = linksCount - 1;

        char* pCharTemp;

        if (partsCount == 0)
            MessageBox("No parts in the model for listing ", "Note!", MB_OK);
        else
        {
            for (int j = 0; j < partsCount ; j++)
                for (int i = 0; i < 25; i++)
                    sLBoxData[j][i] = vParts[j]->name[i];
        }

        int nHere = -1;

        TDlgDialog* listDialog = new TDlgDialog(this, TResID(1003), &nHere);

        for(i = 0; i < 25; i++)
        {
            char append_string[25];
            int append_index = 0;
            while(sLBoxData[i][append_index] != '
            && sLBoxData[i][append_index] != 0)
            {
                append_string[append_index] = sLBoxData[i][append_index++];
            }
            append_string[append_index] = 0;
            strcpy(listDialog->sDialogData[i], append_string);
        }
        if(listDialog->Execute() == IDOK && nHere != -1)
        {
            strcpy(out_string, sLBoxData[nHere]);
            nLBoxIndex = strlen(out_string);
        }
    }
}
Validate(TRUE);
}
int sel_i = listDialog->GetSelection();

if(listDialog->Execute() == IDOK && nHere != -1)
{
  //Find the object with that name
  for (i = 0; i < partsCount; i++)
    if (!strcmp(out_string, vParts[i]->name))
      {
        nCandidPartCount++;
        nCandidPartIndex = i;
      }

  if (nCandidPartCount == 0)
    
    MessageBox("Could not find the part object!",
    "Status Update", MB_OK);
  else
    
    if (nCandidPartCount > 1)
      {
        MessageBox("Found more than one part object with this name!",
        "Status Update", MB_OK);
      }
    else
      
      MessageBox("Found one part object!",
      "Status Update", MB_OK);
  
}

//examine the link_object for directionality
//identify the "from" part_object
//establish pointers in the link and part_objects
//identify the "to" part_object
//establish pointers in the link and part_objects
break;
}
case (1):
{
  MessageBox("Setting Up Blow-Over Link", "Your Selection:", MB_OK);
  BlowOver *blowOver = new BlowOver;
  TEditDialog* editBox = new TEditDialog(this, TRResID(300), &nHere);
  editBox->SetTransferBuffer(editData);
  if(editBox->Execute() == IDOK )
    {
      for (i = 0; i<50; i++)
        blowOver->name[i] = editData[i];
    }
  vLinks[linksCount++] = blowOver;
  break;
}
case (2):
{
    MessageBox("Setting Up Blow-Through Link", "Your Selection:", MB_OK);
    BlowThru *blowThru = new BlowThru;
    TEditDialog* editBox = new TEditDialog(this, TResID(300), &nHere);
    editBox->SetTransferBuffer(editData);
    if(editBox->Execute() == IDOK )
    {
        for (i = 0; i<50; i++)
            blowThru->name[i] = editData[i];
    }
    vLinks[linksCount++] = blowThru;
    break;
}

case (3):
{
    MessageBox("Setting Up Connection Link", "Your Selection:", MB_OK);
    Connection *connection = new Connection;
    TEditDialog* editBox = new TEditDialog(this, TResID(300), &nHere);
    editBox->SetTransferBuffer(editData);
    if(editBox->Execute() == IDOK )
    {
        for (i = 0; i<50; i++)
            connection->name[i] = editData[i];
    }
    vLinks[linksCount++] = connection;
    break;
}

case (4):
{
    MessageBox("Setting up a Feed Link", "Your Selection:", MB_OK);
    Feed *feed = new Feed;
    TEditDialog* editBox = new TEditDialog(this, TResID(300), &nHere);
    editBox->SetTransferBuffer(editData);
    if(editBox->Execute() == IDOK )
    {
        for (i = 0; i<50; i++)
            feed->name[i] = editData[i];
    }
    vLinks[linksCount++] = feed;
    break;
}
default:
{
    MessageBox("Bad Selection, Try Again!", "Your Selection:", MB_OK);
    break;
}
}
void DesMainWin::CmDelRel()
{
}

void DesMainWin::CmModRel()
{
}

void DesMainWin::CmListRel()
{
    // Initialize the List Box items
    if ((partsCount <= 0) || (linksCount <= 0))
        MessageBox("Model is Empty or No links Established Yet.", "Your Selection:", MB_OK);
    else
    {
    int i, j;
    char* pCharTemp;

    if (linksCount == 0)
        MessageBox("Listing Information Case#0", "Your Selection:", MB_OK);
    else
    {
        for (j = 0; j < linksCount; j++)
            for (i = 0; i < 25; i++)
                sLBoxData[j][i] = vLinks[j]->name[i];
    }

    int nHere = -1;

    TDlgDialog* listDialog = new TDlgDialog(this, TResID(3003), &nHere);
    for (i = 0; i < 25; i++)
    {
        char append_string[25];
        int append_index = 0;
        while (sLBoxData[i][append_index] != ' ' && sLBoxData[i][append_index] != 0)
        {
            append_string[append_index] = sLBoxData[i][append_index++];
        }
        append_string[append_index] = 0;
        strcpy(listDialog->sDialogData[i], append_string);
    }

    if (listDialog->Execute() == IDOK && nHere != -1)
    {
        strcpy(out_string, sLBoxData[nHere]);
        nLBoxIndex = strlen(out_string);
        Invalidate(TRUE);
    }

    int selection = listDialog->GetSelection();

    if (listDialog->Execute() == IDOK && nHere != -1)
    {
        switch (selection)
        {
        case 0:
void DesMainWin::CmAddComplaint()
{
    // First: List the existing complaints to prevent duplications

    int i, j;
    char* pCharTemp;

    if ( partsCount == 0 )
    {
        MessageBox("No parts in the model for listing", "Note!", MB_OK);
    }
    else
    {
        int nHere = -1;
        TDlgDialog* listDialog = new TDlgDialog(this, TResID(1003), &nHere);
        for ( i = 0; i < 25; i++ )
listDialog->sDialogData[i][0] = 0;
for(i = 0; i < complaintsCount; i++)
{
    strcpy(listDialog->sDialogData[i], vComplaints[i]->name);
}
listDialog->Execute();

// Second: Get the name of the complaint to be added
nHere = -1;
char editData[25];
editData[0] = 0;
TEditDialog* editBox = new TEditDialog(this, TResID(100), &nHere);
editBox->SetTransferBuffer(editData);
if(editBox->Execute() == IDOK )
{
    vComplaints[complaintsCount] = new Complaint:
        for (i = 0; i < 25; i++)
            vComplaints[complaintsCount]->name[i] = editData[i];
    complaintsCount++;
}

// Third: List the existing faults for the user to choose from
// until he presses OK.

nHere = -1;
j = 0;
do
{
    TDlgDialog* listDialog = new TDlgDialog(this, TResID(1000), &nHere);
    for ( i = 0; i < 25; i++)
        listDialog->sDialogData[i][0] = 0;
    for(int i = 0; i < faultsCount; i++)
    {
        char append_string[25];
        int append_index = 0;
        while(vFaults[i]->name[append_index] != ' ' & & vFaults[i]->name[append_index] != 0)
        {
            append_string[append_index] = vFaults[i]->name[append_index++];
        }
        append_string[append_index] = 0;
        strcpy(listDialog->sDialogData[i], append_string);
    }
    if(listDialog->Execute() == IDOK & & nHere != -1)
    {
        strcpy(vComplaints[complaintsCount-1]->pFaultName[i++],
              vFaults[nHere]->name);
        vComplaints[complaintsCount-1]->nCompFltCounter++;
        Invalidate(TRUE);
    }
    else
        nHere = -1;
}
while ( (j < 5) && (nHere != -1) ):
    
}

void DesMainWin::CmDelComplaint()
{
}

void DesMainWin::CmModComplaint()
{
    // List the existing complaints in the model

    int i, j;
    int selectedComplaint;
    int nCFCounter;
    int selectedFault;
    char* pCharTemp;

    if (partsCount == 0 )
        MessageBox("No parts in the model for listing", "Note!", MB_OK);
    else
        {
        int nHere = -1;
        TDlgDialog* listDialog = new TDlgDialog(this, TResID(1003), &nHere);
        for ( i = 0; i < 25; i++)
            listDialog->sDialogData[i][0] = 0;
        for (i = 0; i < complaintsCount; i++)
        {
            strcpy(listDialog->sDialogData[i], vComplaints[i]->name);
        }

        // Pick the complaint to be modified

        if(listDialog->Execute() == IDOK && nHere != -1)
        {
            selectedComplaint = nHere;
            nCFCounter = vComplaints[selectedComplaint]->nCompFltCounter;
            nHere = -1;
            TDlgDialog* listDialog = new TDlgDialog(this, TResID(1003), &nHere);
            for ( i = 0; i < 25; i++)
                listDialog->sDialogData[i][0] = 0;
            for (int i = 0; i < 5; i++)
            {
                strcpy(listDialog->sDialogData[i],
                        vComplaints[selectedComplaint]->pFaultName[i]);
            }
            if(listDialog->Execute() == IDOK && nHere != -1)
            {
                selectedFault = nHere;
                if (selectedFault < (nCFCounter - 1))
                {
                    for ( i = selectedFault; i < (nCFCounter - 1); i++)
                        strcpy(vComplaints[selectedComplaint]->pFaultName[i],
                                vComplaints[selectedComplaint]->pFaultName[i+1]);
                vComplaints[selectedComplaint]->nCompFltCounter--;
else
{
    if (selectedFault == (nCFCounter - 1))
    {
        vComplaints[selectedComplaint]->pFaultName[selectedFault][0] = 0;
        vComplaints[selectedComplaint]->nCompFltCounter--;
    }
    else
    {
        TDlgDialog* listDialog = new TDlgDialog(this, TResID(1000), &nHere);
        for ( i = 0; i < 25; i++)
            listDialog->sDialogData[i][0] = 0;
        for ( i = 0; i < faultsCount; i++)
        {
            char append_string[25];
            int append_index = 0;
            while(vFaults[i]->name[append_index] != ' ' 
                && vFaults[i]->name[append_index] != 0)
            {
                append_string[append_index] = vFaults[i]->name[append_index++];
            }
            append_string[append_index] = 0;
            strcpy(listDialog->sDialogData[i], append_string);
        }
        if(listDialog->Execute() == IDOK && nHere != -1)
        {
            strcpy(vComplaints[selectedComplaint]->pFaultName[nCFCounter], 
                   vFaults[nHere]->name);
            vComplaints[selectedComplaint]->nCompFltCounter++;
            Invalidate(TRUE);
        }
    }
}

void DesMainWin::CmListComplaints()
{
    int i, j;
    if (partsCount == 0)
        MessageBox("Empty Model! No Complaints to List.", "Warning", MB_OK);
    else
    {
        int nHere = -1;
        TDlgDialog* listDialog = new TDlgDialog(this, TResID(1003), &nHere);
        for ( i = 0; i < 25; i++)
            listDialog->sDialogData[i][0] = 0;
        for(i = 0; i < complaintsCount; i++)
        {
            strcpy(listDialog->sDialogData[i], vComplaints[i]->name);
        }
    }
}
void DesMainWin::CmListFaults()
{
    // Initialize the List Box items
    int i, j;
    if (faultsCount <= 0)
        MessageBox("No Fault Objects to List. "Your Selection:". MB_OK);
    else
    {
        int nHere = -1;
        TDlgDialog* listDialog = new TDlgDialog(this, TResID(3003), &nHere);
        for (i = 0; i < 25; i++)
            listDialog->sDialogData[i][0] = 0;
        for (i = 0; i < faultsCount; i++)
        {
            strcpy(listDialog->sDialogData[i], vFaults[i]->name);
        }
        if (listDialog->Execute() == IDOK && nHere != -1)
        {
            strcpy(out_string, sListBoxData[nHere]);
            nListBoxIndex = strlen(out_string);
            Invalidate(TRUE);
        }
    }
}

void DesMainWin::CmGetComplaint()
{
    // Check if any fault objects exist in the model
    // no diagnostics is allowed without fault objects
    if (faultsCount <= 0)
        MessageBox("No Faults Exist in the Model\n        No Diagnostic Action Allowed!", "Warning", MB_OK);
    else
    {
        int i, j;
        int selectedComplaint, nCFCounter;
        char *pListFaults[5];
        char *pListLinks[5];

        // List the complaints in the model for the user to choose from
        int nHere = -1;
        TDlgDialog* listDialog = new TDlgDialog(this, TResID(1003), &nHere);
        for (i = 0; i < 25; i++)
        {
            listDialog->sDialogData[i][0] = 0;
        }
    }
}
strcpy(listDialog->sDialogData[i]. vComplaints[i]->name);
}
if(listDialog->Execute() == IDOK & nHere != -1)
{
    selectedComplaint = nHere;
    nCFCounter = vComplaints[selectedComplaint]->nCompFltCounter;

    // List the faults under the complaint
    nHere = -1;
    TDlgDialog* listDialog = new TDlgDialog(this. TResID(1003), &nHere);
    for (i = 0; i < 25; i++)
        listDialog->sDialogData[i][0] = 0;
    for (i = 0; i < 5; i++)
    {
        strcpy(listDialog->sDialogData[i],
               vComplaints[selectedComplaint]->pFaultName[i]);
    }
    if(listDialog->Execute() == IDOK & nHere != -1)
    {
        nCandidFaultsIndex = 0;
        nCandidFaultsCount = 0;
        for (i = 0; i < faultsCount; i++)
            if(!strcmp(vComplaints[selectedComplaint]->pFaultName[nHere],
                        vFaults[i]->name))
                {
                nCandidFaultsIndex = i;
                nCandidFaultsCount++;
            }
        if(nCandidFaultsCount == 0)
            MessageBox("No Matching Fault Object!", "Warning Message:", MB_OK);
        else
        {
            if(1 < nCandidFaultsCount)
                MessageBox("Multiple Matching Fault Objects Found\r
" "Last Matching Fault Object was Captured".
"Warning Message:", MB_OK);
        }
    }
}
}

strcpy(out_string, sFaultsLib[nHere]);
nListBoxIndex = strlen(out_string);
Invalidate(TRUE);

int selection = listDialog->GetSelection();

if(listDialog->Execute() == IDOK & nHere != -1)
{
    switch (selection)
    {

case (0):
{
    MessageBoxButton("Processing No-Heat Complaint!", "Your Selection:", MB_OK);

    NoHeat *noHeat = new NoHeat;

    TDlgDialog* listDialog = new TDlgDialog(this, TResID(2000), &nHere);
    for( i = 0; i < 5; i++ )
    {
        strcpy(listDialog->sDialogData[i], noHeat->pFaultName[i]);
    }
    if(listDialog->Execute() == IDOK && nHere != -1)
    {
        strcpy(out_string, noHeat->pFaultName[nHere]);
        nListBoxIndex = strlen(out_string);
        Invalidate(TRUE);
    }
    if( faultsCount <= 0 )
        MessageBoxButton("No Parts in the model!", "Warning Message:", MB_OK);
    else
    {
        nCandidFaultsIndex = 0;
        nCandidFaultsCount = 0;
        for ( i = 0; i < faultsCount; i++ )
        {
            if(strcmp(noHeat->pFaultName[nHere], vFaults[i]->name))
            {
                nCandidFaultsIndex = i;
                nCandidFaultsCount++;
            }
        }
        if(nCandidFaultsCount == 0)
            MessageBoxButton("No Matching Fault Object!", "Warning Message:", MB_OK);
        else
        {
            if( 1 < nCandidFaultsCount)
                MessageBoxButton("Multiple Matching Fault Objects Found \n                 Last Matching Fault Object was Captured", "Warning Message:", MB_OK);
        }
    }*/
 //get the links that each of the faults are related to
 //get the parties in the links for actor, agent, server
 //examine the actor, agent, server for the cause

    break;
}

case (1):
{
    MessageBoxButton("Listing No Air Fault Info!", "Your Selection:", MB_OK);
    NoAir *noAir = new NoAir;

    TDlgDialog* listDialog = new TDlgDialog(this, TResID(2000), &nHere);
    for( i = 0; i < 5; i++ )
    {
strcpy(listDialog->sDialogData[i], noAir->pFaultName[i]);
}
if(listDialog->Execute() == IDOK && nHere != -1)
{
    strcpy(out_string, noAir->pFaultName[nHere]);
    nListBoxIndex = strlen(out_string);
    Invalidate(TRUE);
}
if( faultsCount <= 0 )
    MessageBox("No Parts in the model", "Warning Message:", MB_OK);
else
{
    nCandidFaultsIndex = 0;
    nCandidFaultsCount = 0;
    for( i = 0; i < faultsCount; i++)
        if(!strcmp(noAir->pFaultName[nHere], vFaults[i]->name))
            nCandidFaultsIndex = i;
        nCandidFaultsCount++;
}
if(nCandidFaultsCount == 0)
    MessageBox("No Matching Fault Object!", "Warning Message:", MB_OK);
else
{
    if( 1 < nCandidFaultsCount)
        MessageBox("Multiple Matching Fault Objects Found\n            Last Matching Fault Object was Captured", "Warning Message:", MB_OK);
}
break;
}
    case (2):
{
    MessageBox("Listing No Power Fault Info!", "Your Selection:", MB_OK);
    NoPower *noPower = new NoPower:

TDlgDialog* listDialog = new TDlgDialog(this, TResID(2000), &nHere);
for( i = 0; i < 5; i++)
{
    strcpy(listDialog->sDialogData[i], noPower->pFaultName[i]);
}
if(listDialog->Execute() == IDOK && &nHere != -1)
{
    strcpy(out_string, noAir->pFaultName[nHere]);
    nListBoxIndex = strlen(out_string);
    Invalidate(TRUE);
}
if( faultsCount <= 0 )
    MessageBox("No Parts in the model", "Warning Message:", MB_OK);
else
{
    nCandidFaultsIndex = 0;
nCandidFaultsCount = 0;
for (i = 0; i < faultsCount; i++)
if(!strcmp(noPower-&gt;pFaultName[nHere], vFaults[i]-&gt;name))
{
    nCandidFaultsIndex = i;
    nCandidFaultsCount++;
}
if(nCandidFaultsCount == 0)
    MessageBox("No Matching Fault Object!", "Warning Message:" , MB_OK);
else
{
    if( 1 < nCandidFaultsCount)
    MessageBox("Multiple Matching Fault Objects Found \
    Last Matching Fault Object was Captured", 
    "Warning Message:" , MB_OK);
}
break;
}
default:
{
    MessageBox("Bad Selection, Try Again!","Your Selection:" ,MB_OK);
break;
}
}
}
}
/*

void DesMainWin::CmDispCause()
{
    int i;
    char* pTempChar;
    if ( faultsCount <= 0 )
    MessageBox("No Parts in the model", "Warning Message:" , MB_OK);
else
{
if ( nCandidFaultsIndex < faultsCount )
{
int nHere = -1;
TDlgDialog* listDialog = new TDlgDialog(this, TResID(2001), &amp;nHere);
for ( i = 0; i &lt; 25; i++)
    listDialog-&gt;sDialogData[i][0] = 0;

for( i = 0; i &lt; 5; i++)
{
    char append_string[50];
    int append_index = 0;
    while(vFaults[nCandidFaultsIndex]-&gt;cause[i][append_index] != ' ' 
    &amp;&amp; vFaults[nCandidFaultsIndex]-&gt;cause[i][append_index] != 0)
    {
        append_string[append_index] = 
        vFaults[nCandidFaultsIndex]-&gt;cause[i][append_index++];
}
void DesMainWin::CmDispCondi()
{
    int i;
    if ( faultsCount <= 0 )
        MessageBox("No Parts in the model", "Warning Message:", MB_OK);
    else
    {
        if ( nCandidFaultsIndex < faultsCount )
            int nHere = -1;
        TDlgDialog* listDialog = new TDlgDialog(this, TRslD(2002), &nHere);
        for ( i = 0; i < 25; i++ )
            listDialog->sDialogData[i][0] = 0;

        for ( i = 0; i < 5; i++ )
        {
            char append_string[50];
            int append_index = 0;
            while(vFaults[nCandidFaultsIndex]->condition[i][append_index] != '.'
                   && vFaults[nCandidFaultsIndex]->condition[i][append_index] != 0)
            {
                append_string[append_index] = vFaults[nCandidFaultsIndex]->condition[i][append_index++];
            }
            append_string[append_index] = 0;
            strcpy(listDialog->sDialogData[i], append_string);
        }
        listDialog->Execute();
    }
    else
        MessageBox("No matching fault", "Warning Message:", MB_OK);
}

void DesMainWin::CmDispInitSys()
{
    int i;
    if ( faultsCount <= 0 )
        MessageBox("No Parts in the model", "Warning Message:", MB_OK);
    else
    {
        if ( nCandidFaultsIndex < faultsCount )
        {
            int nHere = -1;
            TDlgDialog* listDialog = new TDlgDialog(this, TRslD(2002), &nHere);
            for ( i = 0; i < 25; i++ )
                listDialog->sDialogData[i][0] = 0;
            for ( i = 0; i < 5; i++ )
            {
                char append_string[50];
                int append_index = 0;
                while(vFaults[nCandidFaultsIndex]->condition[i][append_index] != '.'
                       && vFaults[nCandidFaultsIndex]->condition[i][append_index] != 0)
                {
                    append_string[append_index] = vFaults[nCandidFaultsIndex]->condition[i][append_index++];
                }
                append_string[append_index] = 0;
                strcpy(listDialog->sDialogData[i], append_string);
            }
            listDialog->Execute();
        }
    }
}

int nHere = -1;
TDlgDialog* listDialog = new TDlgDialog(this, TResID(2003), &nHere);
for ( i = 0; i < 25; i++ )
    listDialog->sDialogData[i][0] = 0;
for ( i = 0; i < 5; i-- )
{
    char append_string[50];
    int append_index = 0;
    while(vFaults[nCandidFaultsIndex]->initSys[i][append_index] != ' ' && vFaults[nCandidFaultsIndex]->initSys[i][append_index] != 0)
    {
        append_string[append_index] = vFaults[nCandidFaultsIndex]->initSys[i][append_index++];
    }
    append_string[append_index] = 0;
    strcpy(listDialog->sDialogData[i].append_string);
}
listDialog->Execute();
}
else
    MessageBox("No matching fault", "Warning Message:", MB_OK);
}

void DesMainWin::CmDispInitPart()
{
    int i;
    if ( faultsCount <= 0 )
        MessageBox("No Parts in the model", "Warning Message:", MB_OK);
    else
    {
        if ( nCandidFaultsIndex < faultsCount )
        {
            nHere = -1;
            TDlgDialog* listDialog = new TDlgDialog(this, TResID(2004), &nHere);
            for ( i = 0; i < 25; i++ )
                listDialog->sDialogData[i][0] = 0;
            for ( i = 0; i < 5; i++ )
            {
                char append_string[50];
                int append_index = 0;
                while(vFaults[nCandidFaultsIndex]->initPart[i][append_index] != ' ' && vFaults[nCandidFaultsIndex]->initPart[i][append_index] != 0)
                {
                    append_string[append_index] = vFaults[nCandidFaultsIndex]->initPart[i][append_index++];
                }
                append_string[append_index] = 0;
                strcpy(listDialog->sDialogData[i].append_string);
            }
            listDialog->Execute();
        }
    }
MessageBox("No matching fault", "Warning Message:", MB_OK);
}
}

void DesMainWin::CmDispEffect()
{
    int i;
    if ( faultsCount <= 0 )
        MessageBox("No Parts in the model", "Warning Message:", MB_OK);
    else
    {
        if ( nCandFaultsIndex < faultsCount )
            {
                int nHere = -1;
                TDlgDialog* listDialog = new TDlgDialog(this, TRResID(2005), &nHere);
                for ( i = 0; i < 25; i++ )
                    listDialog->sDialogData[i][0] = 0;

                for ( i = 0; i < 5; i++ )
                    {
                        char append_string[50];
                        int append_index = 0;
                        while(vFaults[nCandFaultsIndex]->effect[i][append_index] != ' '
                                && vFaults[nCandFaultsIndex]->effect[i][append_index] != 0)
                            {
                                append_string[append_index] = vFaults[nCandFaultsIndex]->effect[i][append_index++];
                            }
                        append_string[append_index] = 0;
                        strcpy(listDialog->sDialogData[i], append_string);
                    }
                listDialog->Execute();
            }
    else
        MessageBox("No matching fault", "Warning Message:", MB_OK);
    }
}

void DesMainWin::CmDispSymptom()
{
    int i;
    if ( faultsCount <= 0 )
        MessageBox("No Parts in the model", "Warning Message:", MB_OK);
    else
    {
        if ( nCandFaultsIndex < faultsCount )
            {
                int nHere = -1;
                TDlgDialog* listDialog = new TDlgDialog(this, TRResID(2006), &nHere);
                for ( i = 0; i < 25; i++ )
                    listDialog->sDialogData[i][0] = 0;

                for ( i = 0; i < 5; i++ )
                    {

char append_string[50];
int append_index = 0;
while(vFaults[nCandidFaultsIndex]->symptom[i][append_index] != ' ' 
    && vFaults[nCandidFaultsIndex]->symptom[i][append_index] != 0)
{
    append_string[append_index] = 
        vFaults[nCandidFaultsIndex]->symptom[i][append_index++];
}
append_string[append_index] = 0;
strcpy(listDialog->sDialogData[i], append_string);
}
listDialog->Execute();
else
    MessageBox("No matching fault", "Warning Message:", MB_OK);
;

void DesMainWin::CmDispEffSys()
{
    int i;
    if ( faultsCount <= 0 )
        MessageBox("No Parts in the model", "Warning Message:", MB_OK);
    else
    {
        if ( nCandidFaultsIndex < faultsCount )
        {
            int nHere = -1;
            TDlgDialog* listDialog = new TDlgDialog(this, TRestID(2007), &nHere);
            for ( i = 0; i < 25; i++ )
                listDialog->sDialogData[i][0] = 0;
            for ( i = 0; i < 5; i++ )
            {
                char append_string[50];
                int append_index = 0;
                while(vFaults[nCandidFaultsIndex]->effSys[i][append_index] != ' ' 
                    && vFaults[nCandidFaultsIndex]->effSys[i][append_index] != 0)
                {
                    append_string[append_index] = 
                        vFaults[nCandidFaultsIndex]->effSys[i][append_index++];
                }
                append_string[append_index] = 0;
                strcpy(listDialog->sDialogData[i], append_string);
            }
            listDialog->Execute();
        }
    else
        MessageBox("No matching fault", "Warning Message:", MB_OK);
    }
}

void DesMainWin::CmDispEffPart()
{
int i;
if ( faultsCount <= 0 )
    MessageBox("No Parts in the model", "Warning Message: ", MB_OK);
else
{
    if ( nCandidFaultsIndex < faultsCount )
    {
        nHere = -1;
        TDlgDialog* listDialog = new TDlgDialog(this, TResID(2008), &nHere);
        for ( i = 0; i < 25; i++)
            listDialog->sDialogData[i][0] = 0;

        for ( i = 0; i < 5; i++)
        {
            char append_string[50];
            int append_index = 0;
            while(vFaults[nCandidFaultsIndex]->effPart[i][append_index] != ' ' && vFaults[nCandidFaultsIndex]->effPart[i][append_index] != 0)
                
                append_string[append_index] = vFaults[nCandidFaultsIndex]->effPart[i][append_index++];
            append_string[append_index] = 0;
            strcpy(listDialog->sDialogData[i], append_string);
        }
        listDialog->Execute();
    }
    else
        MessageBox("No matching fault", "Warning Message: ", MB_OK);
}

void DesMainWin::CmDispHowToFix()
{
    int i;
    if ( faultsCount <= 0 )
        MessageBox("No Parts in the model", "Warning Message: ", MB_OK);
    else
    {
        if ( nCandidFaultsIndex < faultsCount )
        {
            nHere = -1;
            TDlgDialog* listDialog = new TDlgDialog(this, TResID(2009), &nHere);
            for ( i = 0; i < 25; i++)
                listDialog->sDialogData[i][0] = 0;

            for ( i = 0; i < 5; i++)
            {
                char append_string[50];
                int append_index = 0;
                while(vFaults[nCandidFaultsIndex]->howToFix[i][append_index] != ' ' && vFaults[nCandidFaultsIndex]->howToFix[i][append_index] != 0)
                {

                    }
append_string[append_index] =
    vFaults[nCandidFaultsIndex].
>
howToFix[i][append_index--];
    }
    append_string[append_index] = 0;
    strcpy(listDialog->sDialogData[i], append_string);
    }
    listDialog->Execute();
    }
    else
    MessageBox("No matching fault", "Warning Message:", MB_OK);
    }
    
    * The Design Module calling other objects
    */

tvoid TTestWindow::HandleButton1Msg()
{
    DesAppl *theApp;
    theApp = new DesAppl();
    theApp->Run();
}

/*
 * File Name: desmenu.h
 * File Content: Definition for the menus in the desing module
 * Constants for the menu
 */
#ifndef _DESMENU_H
#define CM_FILEEXIT 101
#define CM_ADDCOMPONENT 201
#define CM_DELCOMPONENT 202
#define CM_MODCOMPONENT 203
#define CM_LISTCOMPONENT 204
#define CM_SELECTASSEMBLY 301
#define CM_CREATEASSEMBLY 302
#define CM_DELASSEMBLY 303
#define CM_MODASSEMBLY 304
#define CM_LISTASSEMBLY 305
#define CM_LISTALLASSEMBLIES 306
#define CM_SETUPRELP 401
#define CM_DELREL 402
#define CM_MODREL 403
#define CM_LISTREL 404

#define CM_ADDCOMPLAINT 501
#define CM_DELCOMPLAINT 502
#define CM_MODCOMPLAINT 503
#define CM_LISTCOMPLAINTS 504

92
#define CM_LISTFAULTS
#define CM_GETCOMPLAINT
#define CM_DISPCEASE
#define CM_DISPCONDI
#define CM_DISPINITSYS
#define CM_DISPINITPART
#define CM_DISPEFFECT
#define CM_DISPSYMPTOM
#define CM_DISPEFSSYS
#define CM_DISPEFFFART
#define CM_DISPHOWTOFIX

#define _DESMENU_H
#endif

// File: desmenu.h

#include "desmenu.h"

#define DES_PRT_ADD
#define DES_PRT_LIST 1003
#define DES_PRT_NAME 100

#define DIA_COM_PICK
#define DISP_CAUSE
#define DISP_CONDI
#define DISP_INITSYS 2003
#define DISP_INITPART 2004
#define DISP_EFFECT
#define DISP_SYMPOTOM
#define DISP_EFSSYS
#define DISP_EFFFART 2008
#define DISP_HOWTOFIX

#define DES_REL_PICK 3000
#define DES_REL_LIST 3003
#define DES_REL_NAME 300

DESMENU MENU
{

POPUP "&File"
{

MENUITEM "Open ", CM_FILEEXIT
MENUITEM "Save ", CM_FILEEXIT
MENUITEM "Save As", CM_FILEEXIT
MENUITEM "Print ", CM_FILEEXIT
MENUITEM "E&xit ", CM_FILEEXIT

}

POPUP "Parts"
{
MENUITEM "Add", CM_ADDCOMPONENT
MENUITEM "Delete", CM_DELCOMPONENT
MENUITEM "Modify", CM_MODCOMPONENT
MENUITEM "List", CM_LISTCOMPONENT
}

POPUP "Assemblies"
{
MENUITEM "Select", CM_SELECTASSEMBLY
MENUITEM "Create", CM_CREATEASSEMBLY
MENUITEM "Delete", CM_DELASSEMBLY
MENUITEM "Modify", CM_MODASSEMBLY
MENUITEM "List", CM_LISTASSEMBLY
MENUITEM "ListAll", CM_LISTALLASSEMBLIES
}

POPUP "Relations"
{
MENUITEM "Setup", CM_SETUPREL
MENUITEM "Delete", CM_DELREL
MENUITEM "Modify", CM_MODREL
MENUITEM "List", CM_LISTREL
}

POPUP "Complaints"
{
MENUITEM "Add Complaint", CM_ADDCOMPLAINT
MENUITEM "Delete Complaint", CM_DELCOMPLAINT
MENUITEM "Modify Complaint", CM_MODCOMPLAINT
MENUITEM "List Complaints", CM_LISTCOMPLAINTS
MENUITEM "List Faults", CM_LISTFAULTS
}

POPUP "Diagnostics"
{
MENUITEM "Identify Complaints", CM_GETCOMPLAINT
MENUITEM "Disp Cause", CM_DISPENCE
MENUITEM "Disp Condition", CM_DISP_COND
MENUITEM "Disp Init_Syst", CM_DISPINITSYS
MENUITEM "Disp Init_Part", CM_DISPINITPART
MENUITEM "Disp Effect", CM_DISP_EFFECT
MENUITEM "Disp Symptom", CM_DISP_SYMPTOM
MENUITEM "Disp Effected Sys", CM_DISP_EFFSYS
MENUITEM "Disp Effected Part", CM_DISP_EFFPART
MENUITEM "Disp How_To_Fix", CM_DISP_HOWTOFIX
}

DES_PRT_ADD DIALOG 7, 13, 153, 100
STYLE WS_POPUP | WS_DLGFRAME
CAPTION "Add Part"
{
PUSHBUTTON "OK", IDOK, 103, 10, 30, 25
PUSHBUTTON "Cancel", IDCANCEL, 102, 49, 30, 25
CONTROL "LISTBOX", 101, "LISTBOX", LBS_NOTIFY | LBS_SORT | WS_CHILD | WS_VISIBLE | WS_BORDER | WS_CAPTION | WS_VSCROLL, 8, 7, 85, 75
}

DES_PRT_NAME DIALOG 7, 13, 153, 100
STYLE WS_POPUP | WS_DLGFRAME
CAPTION "Type In The Part's Name"
{
    PUSHBUTTON "OK", IDOK, 103, 10, 30, 12
    PUSHBUTTON "Cancel", IDCANCEL, 102, 49, 30, 12
    EDITTEXT 101, 8, 7, 60, 12
}

DES_PRT_LIST DIALOG 7, 13, 160, 100
STYLE WS_POPUP | WS_DLGFRAME
CAPTION "List of Parts in the Model"
{
    PUSHBUTTON "OK", IDOK, 103, 10, 30, 12
    PUSHBUTTON "Cancel", IDCANCEL, 102, 49, 30, 12
    CONTROL "LISTBOX", 101, "LISTBOX", LBS_NOTIFY | LBS_SORT | WS_CHILD | WS_VISIBLE | WS_BORDER | WS_CAPTION | WS_VSCROLL, 8, 7, 85, 75
}

DIA.COM_PICK DIALOG 7, 13, 153, 100
STYLE WS_POPUP | WS_DLGFRAME
CAPTION "Pick One Complaint"
{
    PUSHBUTTON "OK", IDOK, 103, 10, 30, 25
    PUSHBUTTON "Cancel", IDCANCEL, 102, 49, 30, 25
    CONTROL "LISTBOX", 101, "LISTBOX", LBS_NOTIFY | LBS_SORT | WS_CHILD | WS_VISIBLE | WS_BORDER | WS_CAPTION | WS_VSCROLL, 8, 7, 85, 75
}

DISP_CAUSE DIALOG 7, 13, 180, 100
STYLE WS_POPUP | WS_DLGFRAME
CAPTION "Display Fault's Cause"
{
    PUSHBUTTON "OK", IDOK, 75, 80, 30, 12
}

DISP_CONDI DIALOG 7, 13, 180, 100
STYLE WS_POPUP | WS_DLGFRAME
CAPTION "Display Fault's Conditions"
{
    PUSHBUTTON "OK", IDOK, 75, 80, 30, 12
}

DISP_INITSYS DIALOG 7, 13, 180, 100
STYLE WS_POPUP | WS_DLGFRAME
CAPTION "Display Fault's Initiating System"
{
    CONTROL "LISTBOX", 101, "LISTBOX", WS_CHILD | WS_VISIBLE | WS_BORDER |
        WS_VSCROLL, 10, 5, 160, 40
    PUSHBUTTON "OK", IDOK, 75, 80, 30, 12
}

DISP_INITPART DIALOG 7, 13, 180, 100
STYLE WS_POPUP | WS_DLGFRAME
CAPTION "Display Fault's Initiating Part"
{
    CONTROL "LISTBOX", 101, "LISTBOX", WS_CHILD | WS_VISIBLE | WS_BORDER |
        WS_VSCROLL, 10, 5, 160, 40
    PUSHBUTTON "OK", IDOK, 75, 80, 30, 12
}

DISP_EFFECT DIALOG 7, 13, 180, 100
STYLE WS_POPUP | WS_DLGFRAME
CAPTION "Display Fault's Effect"
{
    CONTROL "LISTBOX", 101, "LISTBOX", WS_CHILD | WS_VISIBLE | WS_BORDER |
        WS_VSCROLL, 10, 5, 160, 40
    PUSHBUTTON "OK", IDOK, 75, 80, 30, 12
}

DISP_SYMPTOM DIALOG 7, 13, 180, 100
STYLE WS_POPUP | WS_DLGFRAME
CAPTION "Display Fault's Symptom"
{
    CONTROL "LISTBOX", 101, "LISTBOX", WS_CHILD | WS_VISIBLE | WS_BORDER |
        WS_VSCROLL, 10, 5, 160, 40
    PUSHBUTTON "OK", IDOK, 75, 80, 30, 12
}

DISP_EFFSYS DIALOG 7, 13, 180, 100
STYLE WS_POPUP | WS_DLGFRAME
CAPTION "Display Fault's Effected System"
{
    CONTROL "LISTBOX", 101, "LISTBOX", WS_CHILD | WS_VISIBLE | WS_BORDER |
        WS_VSCROLL, 10, 5, 160, 40
    PUSHBUTTON "OK", IDOK, 75, 80, 30, 12
}

DISP_EFFPART DIALOG 7, 13, 180, 100
STYLE WS_POPUP | WS_DLGFRAME
CAPTION "Display Fault's Effected Part"
{
    CONTROL "LISTBOX", 101, "LISTBOX", WS_CHILD | WS_VISIBLE | WS_BORDER |
        WS_VSCROLL, 10, 5, 160, 40
    PUSHBUTTON "OK", IDOK, 75, 80, 30, 12
}

DISP_HOWTOFIX DIALOG 7, 13, 180, 100
STYLE WS_POPUP | WS_DLGFRAME
CAPTION "Display How to Fix the Fault"
{
CONTROL "LISTBOX", 101, "LISTBOX", WS_CHILD |
WS_VISIBLE | WS_BORDER | WS_VSCROLL, 10, 5, 160, 40
PUSHBUTTON "OK", IDOK, 75, 80, 30, 12
}

DES_REL_PICK DIALOG 7, 13, 153, 100
STYLE WS_POPUP | WS_DLFRAME
CAPTION "Pick The Link Type to Add"
{
PUSHBUTTON "OK", IDOK, 103, 10, 30, 25
PUSHBUTTON "Cancel", IDCANCEL, 102, 49, 30, 25
CONTROL "LISTBOX", 101, "LISTBOX", LBS_NOTIFY | LBS_SORT | WS_CHILD |
WS_VISIBLE | WS_BORDER | WS_CAPTION | WS_VSCROLL, 8, 7, 85, 75
}

DES_REL_LIST DIALOG 7, 13, 160, 100
STYLE WS_POPUP | WS_DLFRAME
CAPTION "List of Links in the Model"
{
PUSHBUTTON "OK", IDOK, 103, 10, 30, 12
PUSHBUTTON "Cancel", IDCANCEL, 102, 49, 30, 12
CONTROL "LISTBOX", 101, "LISTBOX", LBS_NOTIFY | LBS_SORT | WS_CHILD |
WS_VISIBLE | WS_BORDER | WS_CAPTION | WS_VSCROLL, 8, 7, 85, 75
}

DES_REL_NAME DIALOG 7, 13, 153, 100
STYLE WS_POPUP | WS_DLFRAME
CAPTION "Type In The Link's Name"
{
PUSHBUTTON "OK", IDOK, 103, 10, 30, 12
PUSHBUTTON "Cancel", IDCANCEL, 102, 49, 30, 12
EDITTEXT 101, 8, 7, 60, 12
}
Appendix B: C++ Header File for the Fault Class

File Name: Faults.h
Created: By Roozbeh Rahimpour. All Rights Reserved.
Copy Right: December 21, 1996
Description: This file contains class definition for faults expected
in a generic Hair Dryer
Super Class: None
Sub Classes: No Air, No Heat, No Power

/------------------------------ Fault Superclass Definition -------------------

class Fault
{
    public:
        char name[25];
        char cause[5][50];
        char condition[5][50];
        char effect[5][50];
        char symptom[5][50];
        char initSys[5][50];
        char initPart[5][50];
        char effSys[5][50];
        char effPart[5][50];
        char howToFix[5][50];

        Fault();
        virtual ~Fault();
        Fault(const Fault&);
        Fault& operator=(const Fault&);
        int operator==(const Fault& ) const;
        int operator!=(const Fault& ) const;
};

/------------------ Fault Superclass Operator Definitions ------------------

Fault& Fault::operator=(const Fault& ft)
{
    if (this == &ft) return *this;
}

int Fault::operator==(const Fault& ft) const
{
    if (this == &ft) return 1;
}
int Fault::operator!=(const Fault& ft) const
{
    if (this == &ft) return 0;
}

//----------------------- Broken Element Fault Definition -----------------------

class BrokenElement: public Fault
{
    public:

        BrokenElement()
        {
            virtual ~BrokenElement();
        }

        void initBrokenElementFault();
    
    void BrokenElement::initBrokenElementFault()
    {
        int i, j;
        for (i = 0; i < 5; i++)
        {
            cause[i][0] = 0;
            condition[i][0] = 0;
            effect[i][0] = 0;
            symptom[i][0] = 0;
            initSys[i][0] = 0;
            initPart[i][0] = 0;
            effSys[i][0] = 0;
            effPart[i][0] = 0;
            howToFix[i][0] = 0;
        }
        strncpy(name, "Broken Element");
        strncpy(cause[0], "Outside Intrusion through the air duct");
        strncpy(cause[1], "Wear and Tear (rusting)");
        strncpy(condition[0], "Usage under extended period of time");
        strncpy(effect[0], "Disruption in the flow of electrical current");
        strncpy(symptom[0], "Lack of hot air when switch is in position 3");
        strncpy(initSys[0], "Heating System");
        strncpy(initPart[0], "Heating Element");
        strncpy(effSys[0], "Heating System");
        strncpy(effPart[0], "Element");
        strncpy(howToFix[0], "Visually inspect through the air duct to confirm.");
        strncpy(howToFix[1], "Send the unit back to factory if under warranty.");
        strncpy(howToFix[2], "Toss the unit if not under warranty.");
    }

//----------------------- Bad Switch Position Fault Definition ---------------------

class BadSwitchPos: public Fault
{
    public:
void BadSwitchPos::initBadSwitchPosFault()
{
    int i, j;
    for (i = 0; i < 5; i++)
    {
        cause[i][0] = 0;
        condition[i][0] = 0;
        effect[i][0] = 0;
        symptom[i][0] = 0;
        initSys[i][0] = 0;
        initPart[i][0] = 0;
        effSys[i][0] = 0;
        effPart[i][0] = 0;
        howToFix[i][0] = 0;
    }

    strcpy(name, "Bad Switch Position");
    strcpy(cause[0], "Switch in a no contact position");
    strcpy(cause[1], "Extended usage waering down the contacts");
    strcpy(condition[0], " n/a ");
    strcpy(effect[0], "Disruption in the flow of electrical current");
    strcpy(symptom[0], "Lack of air");
    strcpy(symptom[1], "Lack of hot air");
    strcpy(symptom[2], "Interrupted operations; Intermitent on and off");
    strcpy(initSys[0], "Electrical Controls System");
    strcpy(initPart[0], "Switch");
    strcpy(effSys[0], "Heating System");
    strcpy(effSys[1], "Air System");
    strcpy(effPart[0], "Fan");
    strcpy(effPart[1], "Element");
    strcpy(howToFix[0], "Push the switch FIRMLY into position.");
    strcpy(howToFix[1], "If recurring problem, send unit for repair.");
    strcpy(howToFix[2], "Replace the switch.");
}

Auxiliary functions

100

//----------------------- Broken Fan Fault Definition ---------------------

class BrokenFan: public Fault
{
    public:

        BrokenFan(){};
    virtual ~BrokenFan(){};

        void initBrokenFanFault();
};
void BrokenFan::initBrokenFanFault()
{
    int i, j;
    for (i = 0; i < 5; i++)
    {
        cause[i][0] = 0;
        condition[i][0] = 0;
        effect[i][0] = 0;
        symptom[i][0] = 0;
        initSys[i][0] = 0;
        initPart[i][0] = 0;
        effSys[i][0] = 0;
        effPart[i][0] = 0;
        howToFix[i][0] = 0;
    }

    strcpy(name, "Broken Fan");
    strcpy(cause[0], "Worn down brushes");
    strcpy(cause[1], "Exposure to heat under extended usage");
    strcpy(condition[0], "n/a");
    strcpy(effect[0], "Interrupted fan operation");
    strcpy(symptom[0], "No air is blown out of the air duct");
    strcpy(initSys[0], "Air System");
    strcpy(initPart[0], "Fan");
    strcpy(effSys[0], "Air System");
    strcpy(effPart[0], "Fan");
    strcpy(howToFix[0], "Send unit back for repair.");
}

//--------------------- No Power Fault Definition ---------------------

class NoPower: public Fault
{
    public:

        NoPower();
        virtual ~NoPower();

    void initNoPowerFault();
};

void NoPower::initNoPowerFault()
{
    int i, j;
    for (i = 0; i < 5; i++)
    {
        cause[i][0] = 0;
        condition[i][0] = 0;
        effect[i][0] = 0;
        symptom[i][0] = 0;
        initSys[i][0] = 0;
        initPart[i][0] = 0;
        effSys[i][0] = 0;
        effPart[i][0] = 0;
    }
howToFix[i][0] = 0;
}
strcpy(name, "No Power");
strcpy(cause[0], "Black out");
strcpy(cause[1], "Bad switch");
strcpy(cause[2], "Bad fuse");
strcpy(condition[0], "While operating the device");
strcpy(condition[1], "Turning the device on");
strcpy(effect[0], "Can not turn on the device");
strcpy(symptom[0], "The device remains non-operational");
strcpy(initSys[0], "Power System");
strcpy(initPart[0], "n / a");
strcpy(eflSys[0], "Power System");
strcpy(eflSys[1], "Heating System");
strcpy(eflSys[2], "Air System");
strcpy(eflPart[0], "Heating Element");
strcpy(eflPart[1], "Fan");
strcpy(howToFix[0], "Move the switch into pos 2 and 3 a few times.");
strcpy(howToFix[1], "Check the fuse.");
strcpy(howToFix[2], "Check the power cord.");
};

//---------------------------- Bad Fuse Fault Definition ---------------------

class BadFuse: public Fault
{
  public:

    BadFuse(){};
    virtual ~BadFuse(){};
    void initBadFuseFault();
};

void BadFuse::initBadFuseFault()
{
  int i, j;
  for ( i = 0; i < 5; i++)
  {
    cause[i][0] = 0;
    condition[i][0] = 0;
    effect[i][0] = 0;
    symptom[i][0] = 0;
    initSys[i][0] = 0;
    initPart[i][0] = 0;
    eflSys[i][0] = 0;
    eflPart[i][0] = 0;
    howToFix[i][0] = 0;
  }

  strcpy(name, "Bad Fuse");
  strcpy(cause[0], "Wear and Tear");
  strcpy(cause[1], "The unit might have been dropped");
  strcpy(cause[2], "Power Spike");
  strcpy(condition[0], "While operating the device");
  strcpy(effect[0], "Can not turn on the device");
}

102
strcpy(symptom[0], "Device non-operational under any switch position");
strcpy(initSys[0], "Power System");
strcpy(initPart[0], "Fuse");
strcpy(effSys[0], "Power System");
strcpy(effSys[1], "Heating System");
strcpy(effSys[2], "Air System");
strcpy(effPart[0], "Fuse");
strcpy(howToFix[0], "Replace the fuse with a new one.");
strcpy(howToFix[1], "Check device operation after replacing the fuse");

}  

//--------------- Bad Cord Fault Definition ---------------

class BadCord: public Fault
{
public:
    BadCord() {}
    virtual ~BadCord() {}
    void initBadCordFault();
};

void BadCord::initBadCordFault()
{
    int i, j;
    for (i = 0; i < 5; i++)
    {
        cause[i][0] = 0;
        condition[i][0] = 0;
        effect[i][0] = 0;
        symptom[i][0] = 0;
        initSys[i][0] = 0;
        initPart[i][0] = 0;
        effSys[i][0] = 0;
        effPart[i][0] = 0;
        howToFix[i][0] = 0;
    }
    strcpy(name, "Bad Cord");
    strcpy(cause[0], "Sharp bending ");
    strcpy(cause[1], "Factory defect");
    strcpy(condition[0], "While operating the device");
    strcpy(effect[0], "First time usage");
    strcpy(symptom[0], "Device operational intermittently");
    strcpy(initSys[0], "Power System");
    strcpy(initPart[0], "Cord");
    strcpy(effSys[0], "Power System");
    strcpy(effSys[1], "Heating System");
    strcpy(effSys[2], "Air System");
    strcpy(effPart[0], "Cord");
    strcpy(howToFix[0], "Replace cord.");
    strcpy(howToFix[1], "Check device operation after replacing the fuse");
}
Appendix C: C++ Header File for the Component Class

/*
   File Name: parts.h
   Created: By Roozbeh Rahimpour. All Rights Reserved.
   Copyright: December 21, 1996
   Description: This file contains the object declaration requirements
                for a part or component.
   Super Class: None
   Sub Classes: fan, heater, switch, cord, fuse
*/

//------------------------ Super Class Definition ------------------------

class Part
{
  public:
    float  pos_x;
    float  pos_y;
    float  pos_z;
    string colour;
    char   name[25];
    string nextToPartName;
    string *isConnectedTo;
    string *pIsConnectedTo;
    string *pFaults[5];
    string *pRelations[5];

    Part();
    virtual ~Part();
    Part(const Part&);
    Part& operator=(const Part&);
    int operator==(const Part&) const;
    int operator!=(const Part&) const;

    virtual char *isA() {return ("Obj");}
    void move(float delta_x, float delta_y, float delta_z);
    void setName(char *);
    char* getName();
    //    virtual void makeFaultObjects();
};

//------------------------ Super Class Operators ------------------------

Part& Part::operator=(const Part& pt)
{
if (this == &pt) return *this;
}

int Part::operator==(const Part& pt) const
{
    if (this == &pt) return 1;
}

int Part::operator!=(const Part& pt) const
{
    if (this == &pt) return 0;
}

//-------- Super Class Behaviours/Operations/Functions ---------

void Part::move(float delta_x, float delta_y, float delta_z)
{
    pos_x += delta_x;
    pos_y += delta_y;
    pos_z += delta_z;
}

void Part::setName(char* s)
{
    for (int i = 0; i < 50; i++)
        name[i] = s[i];
}

char* Part::getName()
{
    return name;
}

//------------------- Fan Class Definition -------------------------

class Fan: public Part
{
    public:
        float rating;
        float radius;
        string shape;
        string mater;
        float speed;

        Fan();
        virtual ~Fan();

        //
        virtual void makeBrokenFanFaultObject();
    

    //
    /void Fan::makeFaultObjects()
    //{
    //    Fault* brokenFan = new Fault,
    //    DesMainWin::vFaults[DesMainWin::faultsCount++] = brokenFan;

    105
class FanClass
{
    public:
        static Fan* makeFan();
};

Fan* FanClass::makeFan() { return new Fan; }

//--------------------- Element Class Definition ---------------------

class Element: public Part
{
    public:
        int length;
        string shape;
        Fault* brokenElement;

        Element(){
            virtual ~Element();
        }
};

//Element::Element()
//{
//    brokenElement = new Fault;
//    DesMainWin::vFaults[DesMainWin::faultsCount++] = brokenElement;
//}

class ElementClass {
    public:
        static Element* makeElement();
};

Element* ElementClass::makeElement() { return new Element; }

//--------------------- Cord Class Definition ---------------------

class Cord: public Part
{
    public:
        int length;
        string shape;

        Cord(){
            virtual ~Cord();
        }
};

//--------------------- Key (Switch) Class Definition ---------------------

class Key: public Part
{
    public:
        int length;

string shape;
Fault* badSwitchPos;

Key(){
    virtual ~Key(){
};

//Key::Key()
//{
//    badSwitchPos = new Fault;
//    DesMainWin::vFaults[DesMainWin::faultsCount++] = badSwitchPos;
//}

//------------------------ Breaker Class Definition ------------------------

class Breaker: public Part
{
    public:
        int length;
        string shape;

    Breaker(){
        virtual ~Breaker(){
};
Appendix D: C++ Header File for the Relationship Class

```cpp
/*
  File Name: links.h
  Created: By Roozbeh Rahimpour. All Rights Reserved.
          Copy Right: December 21, 1996
  Description: This file defines relationship classes between parts
               and faults
  Super Class: None
  Sub Classes: RelFaults, RelParts

***************************************************************************/

class Link
{
  public:
    char  name[25];
    Part *pLinkFrom;
    Part *pLinkTo;
    Part *pActor;
    Part *pAgent;
    Part *pServer;

    int  priority;
    int  strength;

    Link();
    virtual ~Link();

    Link& operator=(const Link&);
    int operator+=(const Link&) const;
    int operator!=(const Link&) const;

    void setName(char *);
    void setLinkFrom();
    void setLinkTo();
    void setActor();
    void setServer();

    char* getName();
    void getLinkFrom();
    void getLinkTo();
    void getActor();
    void getServer();

};

//--------------- Link Class Operators -----------------------------
```
Link & Link::operator=(const Link & lk) 
{ 
    if (this == &lk) return *this;
}

int Link::operator==(const Link & lk) const 
{ 
    if (this == &lk) return 1;
}

int Link::operator!=(const Link & lk) const 
{ 
    if (this == &lk) return 0;
}

void Link::setName(char* s) 
{ 
    for (int i = 0; i < 50; i++)
        name[i] = s[i];
}

//------------------- Link Class Behaviours and Functions -------------------
char* Link::getName()
{
    return name;
}

//-------------------- OnOff Sub-Class Definition --------------------------
class OnOff: public Link 
{
    public:
        OnOff(){);
            ~OnOff(){);

';

//------------------- BlowOver Sub-Class Definitions -----------------------
class BlowOver: public Link 
{
    public:
        BlowOver(){);
            ~BlowOver(){);

};

//------------------- BlowThru Sub-Class Definitions -----------------------
class BlowThru: public Link 
{
    public:
        BlowThru(){);
            ~BlowThru(){);

};
class Connection: public Link
{
    public:
        Connection();
        ~Connection();

        char *IsA() {return("Connecting Relationship");}
};

class Feed: public Link
{
    public:
        Feed();
        ~Feed();
};
Appendix E: C++ Header File for the Complaint Class

/*******************************************/
File Name: complain.h
Created: By Roozbeh Rahimpour, All Rights Reserved,
Copyright: December 21, 1996
Description: This file defines complaint classes regarding a device
Super Class: None
Sub Classes: RelFaults, RelParts
/*******************************************/

//----------------- Complaint Superclass Definition -----------------

class Complaint
{
   public:
      int nCompFltCounter;
      char name[25];
      char pFaultName[5][25];
      char pPartName[5][25];

      Complaint();
      virtual ~Complaint();
      Complaint(const Complaint&);
      Complaint& operator=(const Complaint&);
      int operator==(const Complaint&) const;
      int operator!=(const Complaint&) const;

};

//----------------- Complaint Superclass Operation Definition -----------------

Complaint& Complaint::operator=(const Complaint& cm)
{
   if (this == &cm) return *this;
}

int Complaint::operator==(const Complaint& cm) const
{
   if (this == &cm) return 1;
}

int Complaint::operator!=(const Complaint& cm) const
{
   if (this == &cm) return 0;
}
// ----------------- No Heat Complaint Class Definition -----------------

class NoHeat: public Complaint
{
    public:
        NoHeat();
        ~NoHeat();
};

NoHeat::NoHeat()
{
    strcpy(name, "No Heat");
    strcpy(pFaultName[0], "Broken Element");
    strcpy(pFaultName[1], "Bad Switch Position");
    pFaultName[2][0] = 0;
    pFaultName[3][0] = 0;
    pFaultName[4][0] = 0;
    nCompFltCounter = 2;
}

// ----------------- No Air Complaint Class Definition -----------------

class NoAir: public Complaint
{
    public:
        NoAir();
        ~NoAir();
};

NoAir::NoAir()
{
    strcpy(name, "No Air");
    strcpy(pFaultName[0], "Broken Fan");
    strcpy(pFaultName[1], "Bad Switch Position");
    pFaultName[2][0] = 0;
    pFaultName[3][0] = 0;
    pFaultName[4][0] = 0;
    nCompFltCounter = 2;
}

// ----------------- No Power Complaint Class Definition -----------------

class NoPower: public Complaint
{
    public:
        NoPower();
        ~NoPower();
};

NoPower::NoPower()
{
    strcpy(name, "No Power");
    strcpy(pFaultName[0], "Bad Cord");
    strcpy(pFaultName[1], "Bad Fuse");
}
pFaultName[2][0] = 0;
pFaultName[3][0] = 0;
pFaultName[4][0] = 0;
nCompFitCounter = 2;
References

3. Editors: Soundar Kumara; Allen L. Soyster; Rangasami L. Kashyap; “Artificial Intelligence (Manufacturing Theory and Practice)”; Institute of Industrial Engineering. 1989, p4-32
31. Watson, I. D. (Editor): Progress in Case-Based Reasoning (Lecture Notes in Artificial Intelligence 1020); Springer-Verlag, Berlin 1995


41. Coyne, R. D., [et al.]: *Knowledge-Based Design Systems*, Addison-Wesley 1990


46. "CAD Gets Objective"; *Byte*, v19 p38 May 1994


VITA AUCTORIS

Roozbeh Rahimpour was born in 1965 in Isfahan, Iran. He immigrated to Canada in 1983 and graduated from H. B. Beal Secondary School (London, Ontario) in 1985. From there he went on to the University of Western Ontario where he obtained a B.E.Sc in Electrical Engineering and a concurrent degree of B.A. in Economics in 1990. He is currently a candidate for the Master’s degree in Applied Science (Industrial and Manufacturing Engineering) and hopes to graduate in Summer 1997.